```c
/*
   +----------------------------------------------------------------------+
   | PHP Version 7                                                        |
   +----------------------------------------------------------------------+
   | Copyright (c) 1997-2018 The PHP Group                                |
   +----------------------------------------------------------------------+
   | This source file is subject to version 3.01 of the PHP license,      |
   | that is bundled with this package in the file LICENSE, and is        |
   | available through the world-wide-web at the following url:           |
   | http://www.php.net/license/3_01.txt                                  |
   | If you did not receive a copy of the PHP license and are unable to   |
   | obtain it through the world-wide-web, please send a note to          |
   | license@php.net so we can mail you a copy immediately.               |
   +----------------------------------------------------------------------+
   | Authors: Marcus Boerger <helly@php.net>                              |
   +----------------------------------------------------------------------+
 */

/* $Id$ */

#ifndef SPL_ARRAY_H
#define SPL_ARRAY_H

#include "php.h"
#include "php_spl.h"
#include "spl_iterators.h"

extern PHPAPI zend_class_entry *spl_ce_ArrayObject;
extern PHPAPI zend_class_entry *spl_ce_ArrayIterator;
extern PHPAPI zend_class_entry *spl_ce_RecursiveArrayIterator;

PHP_MINIT_FUNCTION(spl_array);

extern void spl_array_iterator_append(zval *object, zval *append_value);
extern void spl_array_iterator_key(zval *object, zval *return_value);

#endif /* SPL_ARRAY_H */

/*
 * Local Variables:
 * c-basic-offset: 4
 * tab-width: 4
 * End:
 * vim600: fdm=marker
 * vim: noet sw=4 ts=4
 */
```

```c
/*
   +----------------------------------------------------------------------+
   | PHP Version 7                                                        |
   +----------------------------------------------------------------------+
   | Copyright (c) 1997-2018 The PHP Group                                |
   +----------------------------------------------------------------------+
   | This source file is subject to version 3.01 of the PHP license,      |
   | that is bundled with this package in the file LICENSE, and is        |
   | available through the world-wide-web at the following url:           |
   | http://www.php.net/license/3_01.txt                                  |
   | If you did not receive a copy of the PHP license and are unable to   |
   | obtain it through the world-wide-web, please send a note to          |
   | license@php.net so we can mail you a copy immediately.               |
   +----------------------------------------------------------------------+
   | Authors: Etienne Kneuss <colder@php.net>                             |
   +----------------------------------------------------------------------+
 */

/* $Id$ */

#ifdef HAVE_CONFIG_H
# include "config.h"
#endif

#include "php.h"
#include "zend_exceptions.h"

#include "php_spl.h"
#include "spl_functions.h"
#include "spl_engine.h"
#include "spl_iterators.h"
#include "spl_heap.h"
#include "spl_exceptions.h"

#define PTR_HEAP_BLOCK_SIZE 64

#define SPL_HEAP_CORRUPTED       0x00000001

#define SPL_PQUEUE_EXTR_MASK     0x00000003
#define SPL_PQUEUE_EXTR_BOTH     0x00000003
#define SPL_PQUEUE_EXTR_DATA     0x00000001
#define SPL_PQUEUE_EXTR_PRIORITY 0x00000002

zend_object_handlers spl_handler_SplHeap;
zend_object_handlers spl_handler_SplPriorityQueue;

PHPAPI zend_class_entry  *spl_ce_SplHeap;
PHPAPI zend_class_entry  *spl_ce_SplMaxHeap;
PHPAPI zend_class_entry  *spl_ce_SplMinHeap;
PHPAPI zend_class_entry  *spl_ce_SplPriorityQueue;


typedef void (*spl_ptr_heap_dtor_func)(zval *);
typedef void (*spl_ptr_heap_ctor_func)(zval *);
typedef int  (*spl_ptr_heap_cmp_func)(zval *, zval *, zval *);

typedef struct _spl_ptr_heap {
  zval                   *elements;
  spl_ptr_heap_ctor_func  ctor;
  spl_ptr_heap_dtor_func  dtor;
  spl_ptr_heap_cmp_func   cmp;
  int                     count;
  int                     max_size;
  int                     flags;
} spl_ptr_heap;

typedef struct _spl_heap_object spl_heap_object;
typedef struct _spl_heap_it spl_heap_it;

struct _spl_heap_object {
  spl_ptr_heap       *heap;
  int                 flags;
  zend_class_entry   *ce_get_iterator;
  zend_function      *fptr_cmp;
  zend_function      *fptr_count;
  zend_object         std;
};

/* define an overloaded iterator structure */
struct _spl_heap_it {
  zend_user_iterator  intern;
  int                 flags;
};

static inline spl_heap_object *spl_heap_from_obj(zend_object *obj) /* {{{ */ {
  return (spl_heap_object*)((char*)(obj) - XtOffsetOf(spl_heap_object, std));
}
/* }}} */

#define Z_SPLHEAP_P(zv)  spl_heap_from_obj(Z_OBJ_P((zv)))

static void spl_ptr_heap_zval_dtor(zval *elem) { /* {{{ */
  if (!Z_ISUNDEF_P(elem)) {
    zval_ptr_dtor(elem);
  }
}
/* }}} */

static void spl_ptr_heap_zval_ctor(zval *elem) { /* {{{ */
  Z_TRY_ADDREF_P(elem);
}
/* }}} */

static int spl_ptr_heap_cmp_cb_helper(zval *object, spl_heap_object *heap_object, zval *a, zval *b, zend_long *result) { /* {{{ */
  zval zresult;

  zend_call_method_with_2_params(object, heap_object->std.ce, &heap_object->fptr_cmp, "compare", &zresult, a, b);

  if (EG(exception)) {
    return FAILURE;
  }

  *result = zval_get_long(&zresult);
  zval_ptr_dtor(&zresult);

  return SUCCESS;
}
/* }}} */

static zval *spl_pqueue_extract_helper(zval *value, int flags) /* {{{ */
{
  if ((flags & SPL_PQUEUE_EXTR_BOTH) == SPL_PQUEUE_EXTR_BOTH) {
    return value;
  } else if ((flags & SPL_PQUEUE_EXTR_BOTH) > 0) {
    if ((flags & SPL_PQUEUE_EXTR_DATA) == SPL_PQUEUE_EXTR_DATA) {
      zval *data;
      if ((data = zend_hash_str_find(Z_ARRVAL_P(value), "data", sizeof("data") - 1)) != NULL) {
        return data;
      }
    } else {
      zval *priority;
      if ((priority = zend_hash_str_find(Z_ARRVAL_P(value), "priority", sizeof("priority") - 1)) != NULL) {
        return priority;
      }
    }
  }

  return NULL;
}
/* }}} */

static int spl_ptr_heap_zval_max_cmp(zval *a, zval *b, zval *object) { /* {{{ */
  zval result;

  if (EG(exception)) {
    return 0;
  }

  if (object) {
    spl_heap_object *heap_object = Z_SPLHEAP_P(object);
    if (heap_object->fptr_cmp) {
      zend_long lval = 0;
      if (spl_ptr_heap_cmp_cb_helper(object, heap_object, a, b, &lval) == FAILURE) {
        /* exception or call failure */
        return 0;
      }
      return lval > 0 ? 1 : (lval < 0 ? -1 : 0);
    }
  }

  compare_function(&result, a, b);
  return (int)Z_LVAL(result);
}
/* }}} */

static int spl_ptr_heap_zval_min_cmp(zval *a, zval *b, zval *object) { /* {{{ */
  zval result;

  if (EG(exception)) {
    return 0;
  }

  if (object) {
    spl_heap_object *heap_object = Z_SPLHEAP_P(object);
    if (heap_object->fptr_cmp) {
      zend_long lval = 0;
      if (spl_ptr_heap_cmp_cb_helper(object, heap_object, a, b, &lval) == FAILURE) {
        /* exception or call failure */
        return 0;
      }
      return lval > 0 ? 1 : (lval < 0 ? -1 : 0);
    }
  }

  compare_function(&result, b, a);
  return (int)Z_LVAL(result);
}
/* }}} */

static int spl_ptr_pqueue_zval_cmp(zval *a, zval *b, zval *object) { /* {{{ */
  zval result;
  zval *a_priority_p = spl_pqueue_extract_helper(a, SPL_PQUEUE_EXTR_PRIORITY);
  zval *b_priority_p = spl_pqueue_extract_helper(b, SPL_PQUEUE_EXTR_PRIORITY);

  if ((!a_priority_p) || (!b_priority_p)) {
    zend_error(E_RECOVERABLE_ERROR, "Unable to extract from the PriorityQueue node");
    return 0;
  }

  if (EG(exception)) {
    return 0;
  }

  if (object) {
    spl_heap_object *heap_object = Z_SPLHEAP_P(object);
    if (heap_object->fptr_cmp) {
      zend_long lval = 0;
      if (spl_ptr_heap_cmp_cb_helper((zval *)object, heap_object, a_priority_p, b_priority_p, &lval) == FAILURE) {
        /* exception or call failure */
        return 0;
      }
      return lval > 0 ? 1 : (lval < 0 ? -1 : 0);
    }
  }

  compare_function(&result, a_priority_p, b_priority_p);
  return (int)Z_LVAL(result);
}
/* }}} */

static spl_ptr_heap *spl_ptr_heap_init(spl_ptr_heap_cmp_func cmp, spl_ptr_heap_ctor_func ctor, spl_ptr_heap_dtor_func dtor) /* {{{ */
{
  spl_ptr_heap *heap = emalloc(sizeof(spl_ptr_heap));

  heap->dtor     = dtor;
  heap->ctor     = ctor;
  heap->cmp      = cmp;
  heap->elements = ecalloc(PTR_HEAP_BLOCK_SIZE, sizeof(zval));
  heap->max_size = PTR_HEAP_BLOCK_SIZE;
  heap->count    = 0;
  heap->flags    = 0;

  return heap;
}
/* }}} */

static void spl_ptr_heap_insert(spl_ptr_heap *heap, zval *elem, void *cmp_userdata) { /* {{{ */
  int i;

  if (heap->count+1 > heap->max_size) {
    /* we need to allocate more memory */
    heap->elements  = erealloc(heap->elements, heap->max_size * 2 * sizeof(zval));
    memset(heap->elements + heap->max_size, 0, heap->max_size * sizeof(zval));
    heap->max_size *= 2;
  }

  /* sifting up */
  for (i = heap->count; i > 0 && heap->cmp(&heap->elements[(i-1)/2], elem, cmp_userdata) < 0; i = (i-1)/2) {
    heap->elements[i] = heap->elements[(i-1)/2];
  }
  heap->count++;

  if (EG(exception)) {
    /* exception thrown during comparison */
    heap->flags |= SPL_HEAP_CORRUPTED;
  }

  ZVAL_COPY_VALUE(&heap->elements[i], elem);
}
/* }}} */

static zval *spl_ptr_heap_top(spl_ptr_heap *heap) { /* {{{ */
  if (heap->count == 0) {
    return NULL;
  }

  return Z_ISUNDEF(heap->elements[0])? NULL : &heap->elements[0];
}
/* }}} */

static void spl_ptr_heap_delete_top(spl_ptr_heap *heap, zval *elem, void *cmp_userdata) { /* {{{ */
  int i, j;
  const int limit = (heap->count-1)/2;
  zval *bottom;

  if (heap->count == 0) {
    ZVAL_UNDEF(elem);
    return;
  }

  ZVAL_COPY_VALUE(elem, &heap->elements[0]);
  bottom = &heap->elements[--heap->count];

  for (i = 0; i < limit; i = j) {
    /* Find smaller child */
    j = i * 2 + 1;
    if(j != heap->count && heap->cmp(&heap->elements[j+1], &heap->elements[j], cmp_userdata) > 0) {
      j++; /* next child is bigger */
    }

    /* swap elements between two levels */
    if(heap->cmp(bottom, &heap->elements[j], cmp_userdata) < 0) {
      heap->elements[i] = heap->elements[j];
    } else {
      break;
    }
  }

  if (EG(exception)) {
    /* exception thrown during comparison */
    heap->flags |= SPL_HEAP_CORRUPTED;
  }

  ZVAL_COPY_VALUE(&heap->elements[i], bottom);
}
/* }}} */

static spl_ptr_heap *spl_ptr_heap_clone(spl_ptr_heap *from) { /* {{{ */
  int i;

  spl_ptr_heap *heap = emalloc(sizeof(spl_ptr_heap));

  heap->dtor     = from->dtor;
  heap->ctor     = from->ctor;
  heap->cmp      = from->cmp;
  heap->max_size = from->max_size;
  heap->count    = from->count;
  heap->flags    = from->flags;

  heap->elements = safe_emalloc(sizeof(zval), from->max_size, 0);
  memcpy(heap->elements, from->elements, sizeof(zval)*from->max_size);

  for (i=0; i < heap->count; ++i) {
    heap->ctor(&heap->elements[i]);
  }

  return heap;
}
/* }}} */

static void spl_ptr_heap_destroy(spl_ptr_heap *heap) { /* {{{ */
  int i;

  for (i=0; i < heap->count; ++i) {
    heap->dtor(&heap->elements[i]);
  }

  efree(heap->elements);
  efree(heap);
}
/* }}} */

static int spl_ptr_heap_count(spl_ptr_heap *heap) { /* {{{ */
  return heap->count;
}
/* }}} */

zend_object_iterator *spl_heap_get_iterator(zend_class_entry *ce, zval *object, int by_ref);

static void spl_heap_object_free_storage(zend_object *object) /* {{{ */
{
  spl_heap_object *intern = spl_heap_from_obj(object);

  zend_object_std_dtor(&intern->std);

  spl_ptr_heap_destroy(intern->heap);
}
/* }}} */

static zend_object *spl_heap_object_new_ex(zend_class_entry *class_type, zval *orig, int clone_orig) /* {{{ */
{
  spl_heap_object   *intern;
  zend_class_entry  *parent = class_type;
  int                inherited = 0;

  intern = zend_object_alloc(sizeof(spl_heap_object), parent);

  zend_object_std_init(&intern->std, class_type);
  object_properties_init(&intern->std, class_type);

  intern->flags      = 0;
  intern->fptr_cmp   = NULL;

  if (orig) {
    spl_heap_object *other = Z_SPLHEAP_P(orig);
    intern->ce_get_iterator = other->ce_get_iterator;

    if (clone_orig) {
      intern->heap = spl_ptr_heap_clone(other->heap);
    } else {
      intern->heap = other->heap;
    }

    intern->flags = other->flags;
  } else {
    intern->heap = spl_ptr_heap_init(spl_ptr_heap_zval_max_cmp, spl_ptr_heap_zval_ctor, spl_ptr_heap_zval_dtor);
  }

  intern->std.handlers = &spl_handler_SplHeap;

  while (parent) {
    if (parent == spl_ce_SplPriorityQueue) {
      intern->heap->cmp = spl_ptr_pqueue_zval_cmp;
      intern->flags = SPL_PQUEUE_EXTR_DATA;
      intern->std.handlers = &spl_handler_SplPriorityQueue;
      break;
    }

    if (parent == spl_ce_SplMinHeap) {
      intern->heap->cmp = spl_ptr_heap_zval_min_cmp;
      break;
    }

    if (parent == spl_ce_SplMaxHeap) {
      intern->heap->cmp = spl_ptr_heap_zval_max_cmp;
      break;
    }

    if (parent == spl_ce_SplHeap) {
      break;
    }

    parent = parent->parent;
    inherited = 1;
  }

  if (!parent) { /* this must never happen */
    php_error_docref(NULL, E_COMPILE_ERROR, "Internal compiler error, Class is not child of SplHeap");
  }

  if (inherited) {
    intern->fptr_cmp = zend_hash_str_find_ptr(&class_type->function_table, "compare", sizeof("compare") - 1);
    if (intern->fptr_cmp->common.scope == parent) {
      intern->fptr_cmp = NULL;
    }
    intern->fptr_count = zend_hash_str_find_ptr(&class_type->function_table, "count", sizeof("count") - 1);
    if (intern->fptr_count->common.scope == parent) {
      intern->fptr_count = NULL;
    }
  }

  return &intern->std;
}
/* }}} */

static zend_object *spl_heap_object_new(zend_class_entry *class_type) /* {{{ */
{
  return spl_heap_object_new_ex(class_type, NULL, 0);
}
/* }}} */

static zend_object *spl_heap_object_clone(zval *zobject) /* {{{ */
{
  zend_object        *old_object;
  zend_object        *new_object;

  old_object  = Z_OBJ_P(zobject);
  new_object = spl_heap_object_new_ex(old_object->ce, zobject, 1);

  zend_objects_clone_members(new_object, old_object);

  return new_object;
}
/* }}} */

static int spl_heap_object_count_elements(zval *object, zend_long *count) /* {{{ */
{
  spl_heap_object *intern = Z_SPLHEAP_P(object);

  if (intern->fptr_count) {
    zval rv;
    zend_call_method_with_0_params(object, intern->std.ce, &intern->fptr_count, "count", &rv);
    if (!Z_ISUNDEF(rv)) {
      *count = zval_get_long(&rv);
      zval_ptr_dtor(&rv);
      return SUCCESS;
    }
    *count = 0;
    return FAILURE;
  }

  *count = spl_ptr_heap_count(intern->heap);

  return SUCCESS;
}
/* }}} */

static HashTable* spl_heap_object_get_debug_info_helper(zend_class_entry *ce, zval *obj, int *is_temp) { /* {{{ */
  spl_heap_object *intern = Z_SPLHEAP_P(obj);
  zval tmp, heap_array;
  zend_string *pnstr;
  HashTable *debug_info;
  int  i;

  *is_temp = 1;

  if (!intern->std.properties) {
    rebuild_object_properties(&intern->std);
  }

  debug_info = zend_new_array(zend_hash_num_elements(intern->std.properties) + 1);
  zend_hash_copy(debug_info, intern->std.properties, (copy_ctor_func_t) zval_add_ref);

  pnstr = spl_gen_private_prop_name(ce, "flags", sizeof("flags")-1);
  ZVAL_LONG(&tmp, intern->flags);
  zend_hash_update(debug_info, pnstr, &tmp);
  zend_string_release(pnstr);

  pnstr = spl_gen_private_prop_name(ce, "isCorrupted", sizeof("isCorrupted")-1);
  ZVAL_BOOL(&tmp, intern->heap->flags&SPL_HEAP_CORRUPTED);
  zend_hash_update(debug_info, pnstr, &tmp);
  zend_string_release(pnstr);

  array_init(&heap_array);

  for (i = 0; i < intern->heap->count; ++i) {
    add_index_zval(&heap_array, i, &intern->heap->elements[i]);
    if (Z_REFCOUNTED(intern->heap->elements[i])) {
      Z_ADDREF(intern->heap->elements[i]);
    }
  }

  pnstr = spl_gen_private_prop_name(ce, "heap", sizeof("heap")-1);
  zend_hash_update(debug_info, pnstr, &heap_array);
  zend_string_release(pnstr);

  return debug_info;
}
/* }}} */

static HashTable *spl_heap_object_get_gc(zval *obj, zval **gc_data, int *gc_data_count) /* {{{ */
{
  spl_heap_object *intern = Z_SPLHEAP_P(obj);
  *gc_data = intern->heap->elements;
  *gc_data_count = intern->heap->count;

  return std_object_handlers.get_properties(obj);
}
/* }}} */

static HashTable* spl_heap_object_get_debug_info(zval *obj, int *is_temp) /* {{{ */
{
  return spl_heap_object_get_debug_info_helper(spl_ce_SplHeap, obj, is_temp);
}
/* }}} */

static HashTable* spl_pqueue_object_get_debug_info(zval *obj, int *is_temp) /* {{{ */
{
  return spl_heap_object_get_debug_info_helper(spl_ce_SplPriorityQueue, obj, is_temp);
}
/* }}} */

/* {{{ proto int SplHeap::count()
 Return the number of elements in the heap. */
SPL_METHOD(SplHeap, count)
{
  zend_long count;
  spl_heap_object *intern = Z_SPLHEAP_P(getThis());

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  count = spl_ptr_heap_count(intern->heap);
  RETURN_LONG(count);
}
/* }}} */

/* {{{ proto int SplHeap::isEmpty()
 Return true if the heap is empty. */
SPL_METHOD(SplHeap, isEmpty)
{
  spl_heap_object *intern = Z_SPLHEAP_P(getThis());

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  RETURN_BOOL(spl_ptr_heap_count(intern->heap) == 0);
}
/* }}} */

/* {{{ proto bool SplHeap::insert(mixed value)
     Push $value on the heap */
SPL_METHOD(SplHeap, insert)
{
  zval *value;
  spl_heap_object *intern;

  if (zend_parse_parameters(ZEND_NUM_ARGS(), "z", &value) == FAILURE) {
    return;
  }

  intern = Z_SPLHEAP_P(getThis());

  if (intern->heap->flags & SPL_HEAP_CORRUPTED) {
    zend_throw_exception(spl_ce_RuntimeException, "Heap is corrupted, heap properties are no longer ensured.", 0);
    return;
  }

  Z_TRY_ADDREF_P(value);
  spl_ptr_heap_insert(intern->heap, value, getThis());

  RETURN_TRUE;
}
/* }}} */

/* {{{ proto mixed SplHeap::extract()
     extract the element out of the top of the heap */
SPL_METHOD(SplHeap, extract)
{
  spl_heap_object *intern;

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  intern = Z_SPLHEAP_P(getThis());

  if (intern->heap->flags & SPL_HEAP_CORRUPTED) {
    zend_throw_exception(spl_ce_RuntimeException, "Heap is corrupted, heap properties are no longer ensured.", 0);
    return;
  }

  spl_ptr_heap_delete_top(intern->heap, return_value, getThis());

  if (Z_ISUNDEF_P(return_value)) {
    zend_throw_exception(spl_ce_RuntimeException, "Can't extract from an empty heap", 0);
    return;
  }
}
/* }}} */

/* {{{ proto bool SplPriorityQueue::insert(mixed value, mixed priority)
     Push $value with the priority $priodiry on the priorityqueue */
SPL_METHOD(SplPriorityQueue, insert)
{
  zval *data, *priority, elem;
  spl_heap_object *intern;

  if (zend_parse_parameters(ZEND_NUM_ARGS(), "zz", &data, &priority) == FAILURE) {
    return;
  }

  intern = Z_SPLHEAP_P(getThis());

  if (intern->heap->flags & SPL_HEAP_CORRUPTED) {
    zend_throw_exception(spl_ce_RuntimeException, "Heap is corrupted, heap properties are no longer ensured.", 0);
    return;
  }

  Z_TRY_ADDREF_P(data);
  Z_TRY_ADDREF_P(priority);

  array_init(&elem);
  add_assoc_zval_ex(&elem, "data", sizeof("data") - 1, data);
  add_assoc_zval_ex(&elem, "priority", sizeof("priority") - 1, priority);

  spl_ptr_heap_insert(intern->heap, &elem, getThis());

  RETURN_TRUE;
}
/* }}} */

/* {{{ proto mixed SplPriorityQueue::extract()
     extract the element out of the top of the priority queue */
SPL_METHOD(SplPriorityQueue, extract)
{
  zval value, *value_out;
  spl_heap_object *intern;

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  intern = Z_SPLHEAP_P(getThis());

  if (intern->heap->flags & SPL_HEAP_CORRUPTED) {
    zend_throw_exception(spl_ce_RuntimeException, "Heap is corrupted, heap properties are no longer ensured.", 0);
    return;
  }

  spl_ptr_heap_delete_top(intern->heap, &value, getThis());

  if (Z_ISUNDEF(value)) {
    zend_throw_exception(spl_ce_RuntimeException, "Can't extract from an empty heap", 0);
    return;
  }

  value_out = spl_pqueue_extract_helper(&value, intern->flags);

  if (!value_out) {
    zend_error(E_RECOVERABLE_ERROR, "Unable to extract from the PriorityQueue node");
    zval_ptr_dtor(&value);
    return;
  }

  ZVAL_DEREF(value_out);
  ZVAL_COPY(return_value, value_out);
  zval_ptr_dtor(&value);
}
/* }}} */

/* {{{ proto mixed SplPriorityQueue::top()
     Peek at the top element of the priority queue */
SPL_METHOD(SplPriorityQueue, top)
{
  zval *value, *value_out;
  spl_heap_object *intern;

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  intern = Z_SPLHEAP_P(getThis());

  if (intern->heap->flags & SPL_HEAP_CORRUPTED) {
    zend_throw_exception(spl_ce_RuntimeException, "Heap is corrupted, heap properties are no longer ensured.", 0);
    return;
  }

  value = spl_ptr_heap_top(intern->heap);

  if (!value) {
    zend_throw_exception(spl_ce_RuntimeException, "Can't peek at an empty heap", 0);
    return;
  }

  value_out = spl_pqueue_extract_helper(value, intern->flags);

  if (!value_out) {
    zend_error(E_RECOVERABLE_ERROR, "Unable to extract from the PriorityQueue node");
    return;
  }

  ZVAL_DEREF(value_out);
  ZVAL_COPY(return_value, value_out);
}
/* }}} */


/* {{{ proto int SplPriorityQueue::setExtractFlags(int flags)
 Set the flags of extraction*/
SPL_METHOD(SplPriorityQueue, setExtractFlags)
{
  zend_long value;
  spl_heap_object *intern;

  if (zend_parse_parameters(ZEND_NUM_ARGS(), "l", &value) == FAILURE) {
    return;
  }

  intern = Z_SPLHEAP_P(getThis());

  intern->flags = value & SPL_PQUEUE_EXTR_MASK;

  RETURN_LONG(intern->flags);
}
/* }}} */

/* {{{ proto int SplPriorityQueue::getExtractFlags()
 Get the flags of extraction*/
```

```c
SPL_METHOD(SplPriorityQueue, getExtractFlags)
{
  spl_heap_object *intern;

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  intern = Z_SPLHEAP_P(getThis());

  RETURN_LONG(intern->flags);
}
/* }}} */

/* {{{ proto int SplHeap::recoverFromCorruption()
 Recover from a corrupted state*/
SPL_METHOD(SplHeap, recoverFromCorruption)
{
  spl_heap_object *intern;

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  intern = Z_SPLHEAP_P(getThis());

  intern->heap->flags = intern->heap->flags & ~SPL_HEAP_CORRUPTED;

  RETURN_TRUE;
}
/* }}} */

/* {{{ proto int SplHeap::isCorrupted()
 Tells if the heap is in a corrupted state*/
SPL_METHOD(SplHeap, isCorrupted)
{
  spl_heap_object *intern;

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  intern = Z_SPLHEAP_P(getThis());

  RETURN_BOOL(intern->heap->flags & SPL_HEAP_CORRUPTED);
}
/* }}} */

/* {{{ proto bool SplPriorityQueue::compare(mixed $a, mixed $b)
     compare the priorities */
SPL_METHOD(SplPriorityQueue, compare)
{
  zval *a, *b;

  if (zend_parse_parameters(ZEND_NUM_ARGS(), "zz", &a, &b) == FAILURE) {
    return;
  }

  RETURN_LONG(spl_ptr_heap_zval_max_cmp(a, b, NULL));
}
/* }}} */

/* {{{ proto mixed SplHeap::top()
     Peek at the top element of the heap */
SPL_METHOD(SplHeap, top)
{
  zval *value;
  spl_heap_object *intern;

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  intern = Z_SPLHEAP_P(getThis());

  if (intern->heap->flags & SPL_HEAP_CORRUPTED) {
    zend_throw_exception(spl_ce_RuntimeException, "Heap is corrupted, heap properties are no longer ensured.", 0);
    return;
  }

  value = spl_ptr_heap_top(intern->heap);

  if (!value) {
    zend_throw_exception(spl_ce_RuntimeException, "Can't peek at an empty heap", 0);
    return;
  }

  ZVAL_DEREF(value);
  ZVAL_COPY(return_value, value);
}
/* }}} */

/* {{{ proto bool SplMinHeap::compare(mixed $a, mixed $b)
     compare the values */
SPL_METHOD(SplMinHeap, compare)
{
  zval *a, *b;

  if (zend_parse_parameters(ZEND_NUM_ARGS(), "zz", &a, &b) == FAILURE) {
    return;
  }

  RETURN_LONG(spl_ptr_heap_zval_min_cmp(a, b, NULL));
}
/* }}} */

/* {{{ proto bool SplMaxHeap::compare(mixed $a, mixed $b)
     compare the values */
SPL_METHOD(SplMaxHeap, compare)
{
  zval *a, *b;

  if (zend_parse_parameters(ZEND_NUM_ARGS(), "zz", &a, &b) == FAILURE) {
    return;
  }

  RETURN_LONG(spl_ptr_heap_zval_max_cmp(a, b, NULL));
}
/* }}} */

static void spl_heap_it_dtor(zend_object_iterator *iter) /* {{{ */
{
  spl_heap_it *iterator = (spl_heap_it *)iter;

  zend_user_it_invalidate_current(iter);
  zval_ptr_dtor(&iterator->intern.it.data);
}
/* }}} */

static void spl_heap_it_rewind(zend_object_iterator *iter) /* {{{ */
{
  /* do nothing, the iterator always points to the top element */
}
/* }}} */

static int spl_heap_it_valid(zend_object_iterator *iter) /* {{{ */
{
  return ((Z_SPLHEAP_P(&iter->data))->heap->count != 0 ? SUCCESS : FAILURE);
}
/* }}} */

static zval *spl_heap_it_get_current_data(zend_object_iterator *iter) /* {{{ */
{
  spl_heap_object *object = Z_SPLHEAP_P(&iter->data);
  zval *element = &object->heap->elements[0];

  if (object->heap->flags & SPL_HEAP_CORRUPTED) {
    zend_throw_exception(spl_ce_RuntimeException, "Heap is corrupted, heap properties are no longer ensured.", 0);
    return NULL;
  }

  if (object->heap->count == 0 || Z_ISUNDEF_P(element)) {
    return NULL;
  } else {
    return element;
  }
}
/* }}} */

static zval *spl_pqueue_it_get_current_data(zend_object_iterator *iter) /* {{{ */
{
  spl_heap_object *object = Z_SPLHEAP_P(&iter->data);
  zval *element = &object->heap->elements[0];

  if (object->heap->flags & SPL_HEAP_CORRUPTED) {
    zend_throw_exception(spl_ce_RuntimeException, "Heap is corrupted, heap properties are no longer ensured.", 0);
    return NULL;
  }

  if (object->heap->count == 0 || Z_ISUNDEF_P(element)) {
    return NULL;
  } else {
    zval *data = spl_pqueue_extract_helper(element, object->flags);
    if (!data) {
      zend_error(E_RECOVERABLE_ERROR, "Unable to extract from the PriorityQueue node");
    }
    return data;
  }
}
/* }}} */

static void spl_heap_it_get_current_key(zend_object_iterator *iter, zval *key) /* {{{ */
{
  spl_heap_object *object = Z_SPLHEAP_P(&iter->data);

  ZVAL_LONG(key, object->heap->count - 1);
}
/* }}} */

static void spl_heap_it_move_forward(zend_object_iterator *iter) /* {{{ */
{
  spl_heap_object *object = Z_SPLHEAP_P(&iter->data);
  zval elem;

  if (object->heap->flags & SPL_HEAP_CORRUPTED) {
    zend_throw_exception(spl_ce_RuntimeException, "Heap is corrupted, heap properties are no longer ensured.", 0);
    return;
  }

  spl_ptr_heap_delete_top(object->heap, &elem, &iter->data);

  zval_ptr_dtor(&elem);

  zend_user_it_invalidate_current(iter);
}
/* }}} */

/* {{{  proto int SplHeap::key()
   Return current array key */
SPL_METHOD(SplHeap, key)
{
  spl_heap_object *intern = Z_SPLHEAP_P(getThis());

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  RETURN_LONG(intern->heap->count - 1);
}
/* }}} */

/* {{{ proto void SplHeap::next()
   Move to next entry */
SPL_METHOD(SplHeap, next)
{
  spl_heap_object *intern = Z_SPLHEAP_P(getThis());
  zval elem;
  spl_ptr_heap_delete_top(intern->heap, &elem, getThis());

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  zval_ptr_dtor(&elem);
}
/* }}} */

/* {{{ proto bool SplHeap::valid()
   Check whether the datastructure contains more entries */
SPL_METHOD(SplHeap, valid)
{
  spl_heap_object *intern = Z_SPLHEAP_P(getThis());

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  RETURN_BOOL(intern->heap->count != 0);
}
/* }}} */

/* {{{ proto void SplHeap::rewind()
   Rewind the datastructure back to the start */
SPL_METHOD(SplHeap, rewind)
{
  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }
  /* do nothing, the iterator always points to the top element */
}
/* }}} */

/* {{{ proto mixed|NULL SplHeap::current()
   Return current datastructure entry */
SPL_METHOD(SplHeap, current)
{
  spl_heap_object *intern  = Z_SPLHEAP_P(getThis());
  zval *element = &intern->heap->elements[0];

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  if (!intern->heap->count || Z_ISUNDEF_P(element)) {
    RETURN_NULL();
  } else {
    ZVAL_DEREF(element);
    ZVAL_COPY(return_value, element);
  }
}
/* }}} */

/* {{{ proto mixed|NULL SplPriorityQueue::current()
   Return current datastructure entry */
SPL_METHOD(SplPriorityQueue, current)
{
  spl_heap_object  *intern  = Z_SPLHEAP_P(getThis());
  zval *element = &intern->heap->elements[0];

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  if (!intern->heap->count || Z_ISUNDEF_P(element)) {
    RETURN_NULL();
  } else {
    zval *data = spl_pqueue_extract_helper(element, intern->flags);

    if (!data) {
      zend_error(E_RECOVERABLE_ERROR, "Unable to extract from the PriorityQueue node");
      RETURN_NULL();
    }

    ZVAL_DEREF(data);
    ZVAL_COPY(return_value, data);
  }
}
/* }}} */

/* iterator handler table */
static const zend_object_iterator_funcs spl_heap_it_funcs = {
  spl_heap_it_dtor,
  spl_heap_it_valid,
  spl_heap_it_get_current_data,
  spl_heap_it_get_current_key,
  spl_heap_it_move_forward,
  spl_heap_it_rewind,
  NULL
};

static const zend_object_iterator_funcs spl_pqueue_it_funcs = {
  spl_heap_it_dtor,
  spl_heap_it_valid,
  spl_pqueue_it_get_current_data,
  spl_heap_it_get_current_key,
  spl_heap_it_move_forward,
  spl_heap_it_rewind,
  NULL
};

zend_object_iterator *spl_heap_get_iterator(zend_class_entry *ce, zval *object, int by_ref) /* {{{ */
{
  spl_heap_it     *iterator;
  spl_heap_object *heap_object = Z_SPLHEAP_P(object);

  if (by_ref) {
    zend_throw_exception(spl_ce_RuntimeException, "An iterator cannot be used with foreach by reference", 0);
    return NULL;
  }

  iterator = emalloc(sizeof(spl_heap_it));

  zend_iterator_init(&iterator->intern.it);

  ZVAL_COPY(&iterator->intern.it.data, object);
  iterator->intern.it.funcs = &spl_heap_it_funcs;
  iterator->intern.ce       = ce;
  iterator->flags           = heap_object->flags;
  ZVAL_UNDEF(&iterator->intern.value);

  return &iterator->intern.it;
}
/* }}} */

zend_object_iterator *spl_pqueue_get_iterator(zend_class_entry *ce, zval *object, int by_ref) /* {{{ */
{
  spl_heap_it     *iterator;
  spl_heap_object *heap_object = Z_SPLHEAP_P(object);

  if (by_ref) {
    zend_throw_exception(spl_ce_RuntimeException, "An iterator cannot be used with foreach by reference", 0);
    return NULL;
  }

  iterator = emalloc(sizeof(spl_heap_it));

  zend_iterator_init((zend_object_iterator*)iterator);

  ZVAL_COPY(&iterator->intern.it.data, object);
  iterator->intern.it.funcs = &spl_pqueue_it_funcs;
  iterator->intern.ce       = ce;
  iterator->flags           = heap_object->flags;

  ZVAL_UNDEF(&iterator->intern.value);

  return &iterator->intern.it;
}
/* }}} */

ZEND_BEGIN_ARG_INFO(arginfo_heap_insert, 0)
  ZEND_ARG_INFO(0, value)
ZEND_END_ARG_INFO()

ZEND_BEGIN_ARG_INFO(arginfo_heap_compare, 0)
  ZEND_ARG_INFO(0, a)
  ZEND_ARG_INFO(0, b)
ZEND_END_ARG_INFO()

ZEND_BEGIN_ARG_INFO(arginfo_pqueue_insert, 0)
  ZEND_ARG_INFO(0, value)
  ZEND_ARG_INFO(0, priority)
ZEND_END_ARG_INFO()

ZEND_BEGIN_ARG_INFO(arginfo_pqueue_setflags, 0)
  ZEND_ARG_INFO(0, flags)
ZEND_END_ARG_INFO()

ZEND_BEGIN_ARG_INFO(arginfo_splheap_void, 0)
ZEND_END_ARG_INFO()

static const zend_function_entry spl_funcs_SplMinHeap[] = {
  SPL_ME(SplMinHeap, compare, arginfo_heap_compare, ZEND_ACC_PROTECTED)
  PHP_FE_END
};
static const zend_function_entry spl_funcs_SplMaxHeap[] = {
  SPL_ME(SplMaxHeap, compare, arginfo_heap_compare, ZEND_ACC_PROTECTED)
  PHP_FE_END
};

static const zend_function_entry spl_funcs_SplPriorityQueue[] = {
  SPL_ME(SplPriorityQueue, compare,               arginfo_heap_compare,    ZEND_ACC_PUBLIC)
  SPL_ME(SplPriorityQueue, insert,                arginfo_pqueue_insert,   ZEND_ACC_PUBLIC)
  SPL_ME(SplPriorityQueue, setExtractFlags,       arginfo_pqueue_setflags, ZEND_ACC_PUBLIC)
  SPL_ME(SplPriorityQueue, getExtractFlags,       arginfo_splheap_void,    ZEND_ACC_PUBLIC)
  SPL_ME(SplPriorityQueue, top,                   arginfo_splheap_void,    ZEND_ACC_PUBLIC)
  SPL_ME(SplPriorityQueue, extract,               arginfo_splheap_void,    ZEND_ACC_PUBLIC)
  SPL_ME(SplHeap,          count,                 arginfo_splheap_void,    ZEND_ACC_PUBLIC)
  SPL_ME(SplHeap,          isEmpty,               arginfo_splheap_void,    ZEND_ACC_PUBLIC)
  SPL_ME(SplHeap,          rewind,                arginfo_splheap_void,    ZEND_ACC_PUBLIC)
  SPL_ME(SplPriorityQueue, current,               arginfo_splheap_void,    ZEND_ACC_PUBLIC)
  SPL_ME(SplHeap,          key,                   arginfo_splheap_void,    ZEND_ACC_PUBLIC)
  SPL_ME(SplHeap,          next,                  arginfo_splheap_void,    ZEND_ACC_PUBLIC)
  SPL_ME(SplHeap,          valid,                 arginfo_splheap_void,    ZEND_ACC_PUBLIC)
  SPL_ME(SplHeap,          recoverFromCorruption, arginfo_splheap_void,    ZEND_ACC_PUBLIC)
  SPL_ME(SplHeap,          isCorrupted,           arginfo_splheap_void,    ZEND_ACC_PUBLIC)
  PHP_FE_END
};

static const zend_function_entry spl_funcs_SplHeap[] = {
  SPL_ME(SplHeap, extract,               arginfo_splheap_void, ZEND_ACC_PUBLIC)
  SPL_ME(SplHeap, insert,                arginfo_heap_insert, ZEND_ACC_PUBLIC)
  SPL_ME(SplHeap, top,                   arginfo_splheap_void, ZEND_ACC_PUBLIC)
  SPL_ME(SplHeap, count,                 arginfo_splheap_void, ZEND_ACC_PUBLIC)
  SPL_ME(SplHeap, isEmpty,               arginfo_splheap_void, ZEND_ACC_PUBLIC)
  SPL_ME(SplHeap, rewind,                arginfo_splheap_void, ZEND_ACC_PUBLIC)
  SPL_ME(SplHeap, current,               arginfo_splheap_void, ZEND_ACC_PUBLIC)
  SPL_ME(SplHeap, key,                   arginfo_splheap_void, ZEND_ACC_PUBLIC)
  SPL_ME(SplHeap, next,                  arginfo_splheap_void, ZEND_ACC_PUBLIC)
  SPL_ME(SplHeap, valid,                 arginfo_splheap_void, ZEND_ACC_PUBLIC)
  SPL_ME(SplHeap, recoverFromCorruption, arginfo_splheap_void, ZEND_ACC_PUBLIC)
  SPL_ME(SplHeap, isCorrupted,           arginfo_splheap_void, ZEND_ACC_PUBLIC)
  ZEND_FENTRY(compare, NULL, NULL, ZEND_ACC_PROTECTED|ZEND_ACC_ABSTRACT)
  PHP_FE_END
};
/* }}} */

PHP_MINIT_FUNCTION(spl_heap) /* {{{ */
{
  REGISTER_SPL_STD_CLASS_EX(SplHeap, spl_heap_object_new, spl_funcs_SplHeap);
  memcpy(&spl_handler_SplHeap, zend_get_std_object_handlers(), sizeof(zend_object_handlers));

  spl_handler_SplHeap.offset         = XtOffsetOf(spl_heap_object, std);
  spl_handler_SplHeap.clone_obj      = spl_heap_object_clone;
  spl_handler_SplHeap.count_elements = spl_heap_object_count_elements;
  spl_handler_SplHeap.get_debug_info = spl_heap_object_get_debug_info;
  spl_handler_SplHeap.get_gc         = spl_heap_object_get_gc;
  spl_handler_SplHeap.dtor_obj = zend_objects_destroy_object;
  spl_handler_SplHeap.free_obj = spl_heap_object_free_storage;

  REGISTER_SPL_IMPLEMENTS(SplHeap, Iterator);
  REGISTER_SPL_IMPLEMENTS(SplHeap, Countable);

  spl_ce_SplHeap->get_iterator = spl_heap_get_iterator;

  REGISTER_SPL_SUB_CLASS_EX(SplMinHeap,           SplHeap,        spl_heap_object_new, spl_funcs_SplMinHeap);
  REGISTER_SPL_SUB_CLASS_EX(SplMaxHeap,           SplHeap,        spl_heap_object_new, spl_funcs_SplMaxHeap);

  spl_ce_SplMaxHeap->get_iterator = spl_heap_get_iterator;
  spl_ce_SplMinHeap->get_iterator = spl_heap_get_iterator;

  REGISTER_SPL_STD_CLASS_EX(SplPriorityQueue, spl_heap_object_new, spl_funcs_SplPriorityQueue);
  memcpy(&spl_handler_SplPriorityQueue, zend_get_std_object_handlers(), sizeof(zend_object_handlers));

  spl_handler_SplPriorityQueue.offset         = XtOffsetOf(spl_heap_object, std);
  spl_handler_SplPriorityQueue.clone_obj      = spl_heap_object_clone;
  spl_handler_SplPriorityQueue.count_elements = spl_heap_object_count_elements;
  spl_handler_SplPriorityQueue.get_debug_info = spl_pqueue_object_get_debug_info;
  spl_handler_SplPriorityQueue.get_gc         = spl_heap_object_get_gc;
  spl_handler_SplPriorityQueue.dtor_obj = zend_objects_destroy_object;
  spl_handler_SplPriorityQueue.free_obj = spl_heap_object_free_storage;

  REGISTER_SPL_IMPLEMENTS(SplPriorityQueue, Iterator);
  REGISTER_SPL_IMPLEMENTS(SplPriorityQueue, Countable);

  spl_ce_SplPriorityQueue->get_iterator = spl_pqueue_get_iterator;

  REGISTER_SPL_CLASS_CONST_LONG(SplPriorityQueue, "EXTR_BOTH",      SPL_PQUEUE_EXTR_BOTH);
  REGISTER_SPL_CLASS_CONST_LONG(SplPriorityQueue, "EXTR_PRIORITY",  SPL_PQUEUE_EXTR_PRIORITY);
  REGISTER_SPL_CLASS_CONST_LONG(SplPriorityQueue, "EXTR_DATA",      SPL_PQUEUE_EXTR_DATA);

  return SUCCESS;
}
/* }}} */

/*
 * Local variables:
 * tab-width: 4
 * c-basic-offset: 4
 * End:
 * vim600: fdm=marker
 * vim: noet sw=4 ts=4
 */

```

```c
/*
   +----------------------------------------------------------------------+
   | PHP Version 7                                                        |
   +----------------------------------------------------------------------+
   | Copyright (c) 1997-2018 The PHP Group                                |
   +----------------------------------------------------------------------+
   | This source file is subject to version 3.01 of the PHP license,      |
   | that is bundled with this package in the file LICENSE, and is        |
   | available through the world-wide-web at the following url:           |
   | http://www.php.net/license/3_01.txt                                  |
   | If you did not receive a copy of the PHP license and are unable to   |
   | obtain it through the world-wide-web, please send a note to          |
   | license@php.net so we can mail you a copy immediately.               |
   +----------------------------------------------------------------------+
   | Authors: Marcus Boerger <helly@php.net>                              |
   |          Etienne Kneuss <colder@php.net>                             |
   +----------------------------------------------------------------------+
 */

/* $Id$ */

#ifdef HAVE_CONFIG_H
# include "config.h"
#endif

#include "php.h"
#include "php_ini.h"
#include "ext/standard/info.h"
#include "ext/standard/php_array.h"
#include "ext/standard/php_var.h"
#include "zend_smart_str.h"
#include "zend_interfaces.h"
#include "zend_exceptions.h"

#include "php_spl.h"
#include "spl_functions.h"
#include "spl_engine.h"
#include "spl_observer.h"
#include "spl_iterators.h"
#include "spl_array.h"
#include "spl_exceptions.h"

SPL_METHOD(SplObserver, update);
SPL_METHOD(SplSubject, attach);
SPL_METHOD(SplSubject, detach);
SPL_METHOD(SplSubject, notify);

ZEND_BEGIN_ARG_INFO(arginfo_SplObserver_update, 0)
  ZEND_ARG_OBJ_INFO(0, SplSubject, SplSubject, 0)
ZEND_END_ARG_INFO();

static const zend_function_entry spl_funcs_SplObserver[] = {
  SPL_ABSTRACT_ME(SplObserver, update,   arginfo_SplObserver_update)
  PHP_FE_END
};

ZEND_BEGIN_ARG_INFO(arginfo_SplSubject_attach, 0)
  ZEND_ARG_OBJ_INFO(0, SplObserver, SplObserver, 0)
ZEND_END_ARG_INFO();

ZEND_BEGIN_ARG_INFO(arginfo_SplSubject_void, 0)
ZEND_END_ARG_INFO();

/*ZEND_BEGIN_ARG_INFO_EX(arginfo_SplSubject_notify, 0, 0, 1)
  ZEND_ARG_OBJ_INFO(0, ignore, SplObserver, 1)
ZEND_END_ARG_INFO();*/

static const zend_function_entry spl_funcs_SplSubject[] = {
  SPL_ABSTRACT_ME(SplSubject,  attach,   arginfo_SplSubject_attach)
  SPL_ABSTRACT_ME(SplSubject,  detach,   arginfo_SplSubject_attach)
  SPL_ABSTRACT_ME(SplSubject,  notify,   arginfo_SplSubject_void)
  PHP_FE_END
};

PHPAPI zend_class_entry     *spl_ce_SplObserver;
PHPAPI zend_class_entry     *spl_ce_SplSubject;
PHPAPI zend_class_entry     *spl_ce_SplObjectStorage;
PHPAPI zend_class_entry     *spl_ce_MultipleIterator;

PHPAPI zend_object_handlers spl_handler_SplObjectStorage;

typedef struct _spl_SplObjectStorage { /* {{{ */
  HashTable         storage;
  zend_long         index;
  HashPosition      pos;
  zend_long         flags;
  zend_function    *fptr_get_hash;
  zval             *gcdata;
  size_t            gcdata_num;
  zend_object       std;
} spl_SplObjectStorage; /* }}} */

/* {{{ storage is an assoc array of [zend_object*]=>[zval *obj, zval *inf] */
typedef struct _spl_SplObjectStorageElement {
  zval obj;
  zval inf;
} spl_SplObjectStorageElement; /* }}} */

static inline spl_SplObjectStorage *spl_object_storage_from_obj(zend_object *obj) /* {{{ */ {
  return (spl_SplObjectStorage*)((char*)(obj) - XtOffsetOf(spl_SplObjectStorage, std));
}
/* }}} */

#define Z_SPLOBJSTORAGE_P(zv)  spl_object_storage_from_obj(Z_OBJ_P((zv)))

void spl_SplObjectStorage_free_storage(zend_object *object) /* {{{ */
{
  spl_SplObjectStorage *intern = spl_object_storage_from_obj(object);

  zend_object_std_dtor(&intern->std);

  zend_hash_destroy(&intern->storage);

  if (intern->gcdata != NULL) {
    efree(intern->gcdata);
  }

} /* }}} */

static int spl_object_storage_get_hash(zend_hash_key *key, spl_SplObjectStorage *intern, zval *this, zval *obj) {
  if (intern->fptr_get_hash) {
    zval rv;
    zend_call_method_with_1_params(this, intern->std.ce, &intern->fptr_get_hash, "getHash", &rv, obj);
    if (!Z_ISUNDEF(rv)) {
      if (Z_TYPE(rv) == IS_STRING) {
        key->key = Z_STR(rv);
        return SUCCESS;
      } else {
        zend_throw_exception(spl_ce_RuntimeException, "Hash needs to be a string", 0);

        zval_ptr_dtor(&rv);
        return FAILURE;
      }
    } else {
      return FAILURE;
    }
  } else {
    key->key = NULL;
    key->h = Z_OBJ_HANDLE_P(obj);
    return SUCCESS;
  }
}

static void spl_object_storage_free_hash(spl_SplObjectStorage *intern, zend_hash_key *key) {
  if (key->key) {
    zend_string_release(key->key);
  }
}

static void spl_object_storage_dtor(zval *element) /* {{{ */
{
  spl_SplObjectStorageElement *el = Z_PTR_P(element);
  zval_ptr_dtor(&el->obj);
  zval_ptr_dtor(&el->inf);
  efree(el);
} /* }}} */

static spl_SplObjectStorageElement* spl_object_storage_get(spl_SplObjectStorage *intern, zend_hash_key *key) /* {{{ */
{
  if (key->key) {
    return zend_hash_find_ptr(&intern->storage, key->key);
  } else {
    return zend_hash_index_find_ptr(&intern->storage, key->h);
  }
} /* }}} */

spl_SplObjectStorageElement *spl_object_storage_attach(spl_SplObjectStorage *intern, zval *this, zval *obj, zval *inf) /* {{{ */
{
  spl_SplObjectStorageElement *pelement, element;
  zend_hash_key key;
  if (spl_object_storage_get_hash(&key, intern, this, obj) == FAILURE) {
    return NULL;
  }

  pelement = spl_object_storage_get(intern, &key);

  if (pelement) {
    zval_ptr_dtor(&pelement->inf);
    if (inf) {
      ZVAL_COPY(&pelement->inf, inf);
    } else {
      ZVAL_NULL(&pelement->inf);
    }
    spl_object_storage_free_hash(intern, &key);
    return pelement;
  }

  ZVAL_COPY(&element.obj, obj);
  if (inf) {
    ZVAL_COPY(&element.inf, inf);
  } else {
    ZVAL_NULL(&element.inf);
  }
  if (key.key) {
    pelement = zend_hash_update_mem(&intern->storage, key.key, &element, sizeof(spl_SplObjectStorageElement));
  } else {
    pelement = zend_hash_index_update_mem(&intern->storage, key.h, &element, sizeof(spl_SplObjectStorageElement));
  }
  spl_object_storage_free_hash(intern, &key);
  return pelement;
} /* }}} */

static int spl_object_storage_detach(spl_SplObjectStorage *intern, zval *this, zval *obj) /* {{{ */
{
  int ret = FAILURE;
  zend_hash_key key;
  if (spl_object_storage_get_hash(&key, intern, this, obj) == FAILURE) {
    return ret;
  }
  if (key.key) {
    ret = zend_hash_del(&intern->storage, key.key);
  } else {
    ret = zend_hash_index_del(&intern->storage, key.h);
  }
  spl_object_storage_free_hash(intern, &key);

  return ret;
} /* }}}*/

void spl_object_storage_addall(spl_SplObjectStorage *intern, zval *this, spl_SplObjectStorage *other) { /* {{{ */
  spl_SplObjectStorageElement *element;

  ZEND_HASH_FOREACH_PTR(&other->storage, element) {
    spl_object_storage_attach(intern, this, &element->obj, &element->inf);
  } ZEND_HASH_FOREACH_END();

  intern->index = 0;
} /* }}} */

static zend_object *spl_object_storage_new_ex(zend_class_entry *class_type, zval *orig) /* {{{ */
{
  spl_SplObjectStorage *intern;
  zend_class_entry *parent = class_type;

  intern = emalloc(sizeof(spl_SplObjectStorage) + zend_object_properties_size(parent));
  memset(intern, 0, sizeof(spl_SplObjectStorage) - sizeof(zval));
  intern->pos = HT_INVALID_IDX;

  zend_object_std_init(&intern->std, class_type);
  object_properties_init(&intern->std, class_type);

  zend_hash_init(&intern->storage, 0, NULL, spl_object_storage_dtor, 0);

  intern->std.handlers = &spl_handler_SplObjectStorage;

  while (parent) {
    if (parent == spl_ce_SplObjectStorage) {
      if (class_type != spl_ce_SplObjectStorage) {
        intern->fptr_get_hash = zend_hash_str_find_ptr(&class_type->function_table, "gethash", sizeof("gethash") - 1);
        if (intern->fptr_get_hash->common.scope == spl_ce_SplObjectStorage) {
          intern->fptr_get_hash = NULL;
        }
      }
      break;
    }

    parent = parent->parent;
  }

  if (orig) {
    spl_SplObjectStorage *other = Z_SPLOBJSTORAGE_P(orig);
    spl_object_storage_addall(intern, orig, other);
  }

  return &intern->std;
}
/* }}} */

/* {{{ spl_object_storage_clone */
static zend_object *spl_object_storage_clone(zval *zobject)
{
  zend_object *old_object;
  zend_object *new_object;

  old_object = Z_OBJ_P(zobject);
  new_object = spl_object_storage_new_ex(old_object->ce, zobject);

  zend_objects_clone_members(new_object, old_object);

  return new_object;
}
/* }}} */

static HashTable* spl_object_storage_debug_info(zval *obj, int *is_temp) /* {{{ */
{
  spl_SplObjectStorage *intern = Z_SPLOBJSTORAGE_P(obj);
  spl_SplObjectStorageElement *element;
  HashTable *props;
  zval tmp, storage;
  zend_string *md5str;
  zend_string *zname;
  HashTable *debug_info;

  *is_temp = 1;

  props = Z_OBJPROP_P(obj);

  debug_info = zend_new_array(zend_hash_num_elements(props) + 1);
  zend_hash_copy(debug_info, props, (copy_ctor_func_t)zval_add_ref);

  array_init(&storage);

  ZEND_HASH_FOREACH_PTR(&intern->storage, element) {
    md5str = php_spl_object_hash(&element->obj);
    array_init(&tmp);
    /* Incrementing the refcount of obj and inf would confuse the garbage collector.
     * Prefer to null the destructor */
    Z_ARRVAL_P(&tmp)->pDestructor = NULL;
    add_assoc_zval_ex(&tmp, "obj", sizeof("obj") - 1, &element->obj);
    add_assoc_zval_ex(&tmp, "inf", sizeof("inf") - 1, &element->inf);
    zend_hash_update(Z_ARRVAL(storage), md5str, &tmp);
    zend_string_release(md5str);
  } ZEND_HASH_FOREACH_END();

  zname = spl_gen_private_prop_name(spl_ce_SplObjectStorage, "storage", sizeof("storage")-1);
  zend_symtable_update(debug_info, zname, &storage);
  zend_string_release(zname);

  return debug_info;
}
/* }}} */

/* overriden for garbage collection */
static HashTable *spl_object_storage_get_gc(zval *obj, zval **table, int *n) /* {{{ */
{
  int i = 0;
  spl_SplObjectStorage *intern = Z_SPLOBJSTORAGE_P(obj);
  spl_SplObjectStorageElement *element;

  if (intern->storage.nNumOfElements * 2 > intern->gcdata_num) {
    intern->gcdata_num = intern->storage.nNumOfElements * 2;
    intern->gcdata = (zval*)erealloc(intern->gcdata, sizeof(zval) * intern->gcdata_num);
  }

  ZEND_HASH_FOREACH_PTR(&intern->storage, element) {
    ZVAL_COPY_VALUE(&intern->gcdata[i++], &element->obj);
    ZVAL_COPY_VALUE(&intern->gcdata[i++], &element->inf);
  } ZEND_HASH_FOREACH_END();

  *table = intern->gcdata;
  *n = i;

  return std_object_handlers.get_properties(obj);
}
/* }}} */

static int spl_object_storage_compare_info(zval *e1, zval *e2) /* {{{ */
{
  spl_SplObjectStorageElement *s1 = (spl_SplObjectStorageElement*)Z_PTR_P(e1);
  spl_SplObjectStorageElement *s2 = (spl_SplObjectStorageElement*)Z_PTR_P(e2);
  zval result;

  if (compare_function(&result, &s1->inf, &s2->inf) == FAILURE) {
    return 1;
  }

  return Z_LVAL(result) > 0 ? 1 : (Z_LVAL(result) < 0 ? -1 : 0);
}
/* }}} */

static int spl_object_storage_compare_objects(zval *o1, zval *o2) /* {{{ */
{
  zend_object *zo1 = (zend_object *)Z_OBJ_P(o1);
  zend_object *zo2 = (zend_object *)Z_OBJ_P(o2);

  if (zo1->ce != spl_ce_SplObjectStorage || zo2->ce != spl_ce_SplObjectStorage) {
    return 1;
  }

  return zend_hash_compare(&(Z_SPLOBJSTORAGE_P(o1))->storage, &(Z_SPLOBJSTORAGE_P(o2))->storage, (compare_func_t)spl_object_storage_compare_info, 0);
}
/* }}} */

/* {{{ spl_array_object_new */
static zend_object *spl_SplObjectStorage_new(zend_class_entry *class_type)
{
  return spl_object_storage_new_ex(class_type, NULL);
}
/* }}} */

int spl_object_storage_contains(spl_SplObjectStorage *intern, zval *this, zval *obj) /* {{{ */
{
  int found;
  zend_hash_key key;
  if (spl_object_storage_get_hash(&key, intern, this, obj) == FAILURE) {
    return 0;
  }

  if (key.key) {
    found = zend_hash_exists(&intern->storage, key.key);
  } else {
    found = zend_hash_index_exists(&intern->storage, key.h);
  }
  spl_object_storage_free_hash(intern, &key);
  return found;
} /* }}} */

/* {{{ proto void SplObjectStorage::attach(object obj, mixed inf = NULL)
 Attaches an object to the storage if not yet contained */
SPL_METHOD(SplObjectStorage, attach)
{
  zval *obj, *inf = NULL;

  spl_SplObjectStorage *intern = Z_SPLOBJSTORAGE_P(getThis());

  if (zend_parse_parameters(ZEND_NUM_ARGS(), "o|z!", &obj, &inf) == FAILURE) {
    return;
  }
  spl_object_storage_attach(intern, getThis(), obj, inf);
} /* }}} */

/* {{{ proto void SplObjectStorage::detach(object obj)
 Detaches an object from the storage */
SPL_METHOD(SplObjectStorage, detach)
{
  zval *obj;
  spl_SplObjectStorage *intern = Z_SPLOBJSTORAGE_P(getThis());

  if (zend_parse_parameters(ZEND_NUM_ARGS(), "o", &obj) == FAILURE) {
    return;
  }
  spl_object_storage_detach(intern, getThis(), obj);

  zend_hash_internal_pointer_reset_ex(&intern->storage, &intern->pos);
  intern->index = 0;
} /* }}} */

/* {{{ proto string SplObjectStorage::getHash(object obj)
 Returns the hash of an object */
SPL_METHOD(SplObjectStorage, getHash)
{
  zval *obj;

  if (zend_parse_parameters(ZEND_NUM_ARGS(), "o", &obj) == FAILURE) {
    return;
  }

  RETURN_NEW_STR(php_spl_object_hash(obj));

} /* }}} */

/* {{{ proto mixed SplObjectStorage::offsetGet(object obj)
 Returns associated information for a stored object */
SPL_METHOD(SplObjectStorage, offsetGet)
{
  zval *obj;
  spl_SplObjectStorageElement *element;
  spl_SplObjectStorage *intern = Z_SPLOBJSTORAGE_P(getThis());
  zend_hash_key key;

  if (zend_parse_parameters(ZEND_NUM_ARGS(), "o", &obj) == FAILURE) {
    return;
  }

  if (spl_object_storage_get_hash(&key, intern, getThis(), obj) == FAILURE) {
    return;
  }

  element = spl_object_storage_get(intern, &key);
  spl_object_storage_free_hash(intern, &key);

  if (!element) {
    zend_throw_exception_ex(spl_ce_UnexpectedValueException, 0, "Object not found");
  } else {
    zval *value = &element->inf;

    ZVAL_DEREF(value);
    ZVAL_COPY(return_value, value);
  }
} /* }}} */

/* {{{ proto bool SplObjectStorage::addAll(SplObjectStorage $os)
 Add all elements contained in $os */
SPL_METHOD(SplObjectStorage, addAll)
{
  zval *obj;
  spl_SplObjectStorage *intern = Z_SPLOBJSTORAGE_P(getThis());
  spl_SplObjectStorage *other;

  if (zend_parse_parameters(ZEND_NUM_ARGS(), "O", &obj, spl_ce_SplObjectStorage) == FAILURE) {
    return;
  }

  other = Z_SPLOBJSTORAGE_P(obj);

  spl_object_storage_addall(intern, getThis(), other);

  RETURN_LONG(zend_hash_num_elements(&intern->storage));
} /* }}} */

/* {{{ proto bool SplObjectStorage::removeAll(SplObjectStorage $os)
 Remove all elements contained in $os */
SPL_METHOD(SplObjectStorage, removeAll)
{
  zval *obj;
  spl_SplObjectStorage *intern = Z_SPLOBJSTORAGE_P(getThis());
  spl_SplObjectStorage *other;
  spl_SplObjectStorageElement *element;

  if (zend_parse_parameters(ZEND_NUM_ARGS(), "O", &obj, spl_ce_SplObjectStorage) == FAILURE) {
    return;
  }

  other = Z_SPLOBJSTORAGE_P(obj);

  zend_hash_internal_pointer_reset(&other->storage);
  while ((element = zend_hash_get_current_data_ptr(&other->storage)) != NULL) {
    if (spl_object_storage_detach(intern, getThis(), &element->obj) == FAILURE) {
      zend_hash_move_forward(&other->storage);
    }
  }

  zend_hash_internal_pointer_reset_ex(&intern->storage, &intern->pos);
  intern->index = 0;

  RETURN_LONG(zend_hash_num_elements(&intern->storage));
} /* }}} */

/* {{{ proto bool SplObjectStorage::removeAllExcept(SplObjectStorage $os)
 Remove elements not common to both this SplObjectStorage instance and $os */
SPL_METHOD(SplObjectStorage, removeAllExcept)
{
  zval *obj;
  spl_SplObjectStorage *intern = Z_SPLOBJSTORAGE_P(getThis());
  spl_SplObjectStorage *other;
  spl_SplObjectStorageElement *element;

  if (zend_parse_parameters(ZEND_NUM_ARGS(), "O", &obj, spl_ce_SplObjectStorage) == FAILURE) {
    return;
  }

  other = Z_SPLOBJSTORAGE_P(obj);

  ZEND_HASH_FOREACH_PTR(&intern->storage, element) {
    if (!spl_object_storage_contains(other, getThis(), &element->obj)) {
      spl_object_storage_detach(intern, getThis(), &element->obj);
    }
  } ZEND_HASH_FOREACH_END();

  zend_hash_internal_pointer_reset_ex(&intern->storage, &intern->pos);
  intern->index = 0;

  RETURN_LONG(zend_hash_num_elements(&intern->storage));
}
/* }}} */

/* {{{ proto bool SplObjectStorage::contains(object obj)
 Determine whethe an object is contained in the storage */
SPL_METHOD(SplObjectStorage, contains)
{
  zval *obj;
  spl_SplObjectStorage *intern = Z_SPLOBJSTORAGE_P(getThis());

  if (zend_parse_parameters(ZEND_NUM_ARGS(), "o", &obj) == FAILURE) {
    return;
  }
  RETURN_BOOL(spl_object_storage_contains(intern, getThis(), obj));
} /* }}} */

/* {{{ proto int SplObjectStorage::count()
 Determine number of objects in storage */
SPL_METHOD(SplObjectStorage, count)
{
  spl_SplObjectStorage *intern = Z_SPLOBJSTORAGE_P(getThis());
  zend_long mode = COUNT_NORMAL;

  if (zend_parse_parameters(ZEND_NUM_ARGS(), "|l", &mode) == FAILURE) {
    return;
  }

  if (mode == COUNT_RECURSIVE) {
    zend_long ret;

    if (mode != COUNT_RECURSIVE) {
      ret = zend_hash_num_elements(&intern->storage);
    } else {
      ret = php_count_recursive(&intern->storage);
    }

    RETURN_LONG(ret);
    return;
  }

  RETURN_LONG(zend_hash_num_elements(&intern->storage));
} /* }}} */

/* {{{ proto void SplObjectStorage::rewind()
 Rewind to first position */
SPL_METHOD(SplObjectStorage, rewind)
{
  spl_SplObjectStorage *intern = Z_SPLOBJSTORAGE_P(getThis());

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  zend_hash_internal_pointer_reset_ex(&intern->storage, &intern->pos);
  intern->index = 0;
} /* }}} */

/* {{{ proto bool SplObjectStorage::valid()
 Returns whether current position is valid */
SPL_METHOD(SplObjectStorage, valid)
{
  spl_SplObjectStorage *intern = Z_SPLOBJSTORAGE_P(getThis());

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  RETURN_BOOL(zend_hash_has_more_elements_ex(&intern->storage, &intern->pos) == SUCCESS);
} /* }}} */

/* {{{ proto mixed SplObjectStorage::key()
 Returns current key */
SPL_METHOD(SplObjectStorage, key)
{
  spl_SplObjectStorage *intern = Z_SPLOBJSTORAGE_P(getThis());

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  RETURN_LONG(intern->index);
} /* }}} */

/* {{{ proto mixed SplObjectStorage::current()
 Returns current element */
SPL_METHOD(SplObjectStorage, current)
{
  spl_SplObjectStorageElement *element;
  spl_SplObjectStorage *intern = Z_SPLOBJSTORAGE_P(getThis());

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  if ((element = zend_hash_get_current_data_ptr_ex(&intern->storage, &intern->pos)) == NULL) {
    return;
  }
  ZVAL_COPY(return_value, &element->obj);
} /* }}} */

/* {{{ proto mixed SplObjectStorage::getInfo()
 Returns associated information to current element */
SPL_METHOD(SplObjectStorage, getInfo)
{
  spl_SplObjectStorageElement *element;
  spl_SplObjectStorage *intern = Z_SPLOBJSTORAGE_P(getThis());

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  if ((element = zend_hash_get_current_data_ptr_ex(&intern->storage, &intern->pos)) == NULL) {
    return;
  }
  ZVAL_COPY(return_value, &element->inf);
} /* }}} */

/* {{{ proto mixed SplObjectStorage::setInfo(mixed $inf)
 Sets associated information of current element to $inf */
SPL_METHOD(SplObjectStorage, setInfo)
{
  spl_SplObjectStorageElement *element;
  spl_SplObjectStorage *intern = Z_SPLOBJSTORAGE_P(getThis());
  zval *inf;

  if (zend_parse_parameters(ZEND_NUM_ARGS(), "z", &inf) == FAILURE) {
    return;
  }

  if ((element = zend_hash_get_current_data_ptr_ex(&intern->storage, &intern->pos)) == NULL) {
    return;
  }
  zval_ptr_dtor(&element->inf);
  ZVAL_COPY(&element->inf, inf);
} /* }}} */

/* {{{ proto void SplObjectStorage::next()
 Moves position forward */
SPL_METHOD(SplObjectStorage, next)
{
  spl_SplObjectStorage *intern = Z_SPLOBJSTORAGE_P(getThis());

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  zend_hash_move_forward_ex(&intern->storage, &intern->pos);
  intern->index++;
} /* }}} */

/* {{{ proto string SplObjectStorage::serialize()
 Serializes storage */
SPL_METHOD(SplObjectStorage, serialize)
{
  spl_SplObjectStorage *intern = Z_SPLOBJSTORAGE_P(getThis());

  spl_SplObjectStorageElement *element;
  zval members, flags;
  HashPosition      pos;
  php_serialize_data_t var_hash;
  smart_str buf = {0};

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  PHP_VAR_SERIALIZE_INIT(var_hash);

  /* storage */
  smart_str_appendl(&buf, "x:", 2);
  ZVAL_LONG(&flags, zend_hash_num_elements(&intern->storage));
  php_var_serialize(&buf, &flags, &var_hash);
  zval_ptr_dtor(&flags);

  zend_hash_internal_pointer_reset_ex(&intern->storage, &pos);

  while (zend_hash_has_more_elements_ex(&intern->storage, &pos) == SUCCESS) {
    if ((element = zend_hash_get_current_data_ptr_ex(&intern->storage, &pos)) == NULL) {
      smart_str_free(&buf);
      PHP_VAR_SERIALIZE_DESTROY(var_hash);
      RETURN_NULL();
    }
    php_var_serialize(&buf, &element->obj, &var_hash);
    smart_str_appendc(&buf, ',');
    php_var_serialize(&buf, &element->inf, &var_hash);
    smart_str_appendc(&buf, ';');
    zend_hash_move_forward_ex(&intern->storage, &pos);
  }

  /* members */
  smart_str_appendl(&buf, "m:", 2);

  ZVAL_ARR(&members, zend_array_dup(zend_std_get_properties(getThis())));
  php_var_serialize(&buf, &members, &var_hash); /* finishes the string */
  zval_ptr_dtor(&members);

  /* done */
  PHP_VAR_SERIALIZE_DESTROY(var_hash);

  if (buf.s) {
    RETURN_NEW_STR(buf.s);
  } else {
    RETURN_NULL();
  }

} /* }}} */

/* {{{ proto void SplObjectStorage::unserialize(string serialized)
 Unserializes storage */
```

```c
SPL_METHOD(SplObjectStorage, unserialize)
{
  spl_SplObjectStorage *intern = Z_SPLOBJSTORAGE_P(getThis());

  char *buf;
  size_t buf_len;
  const unsigned char *p, *s;
  php_unserialize_data_t var_hash;
  zval entry, inf;
  zval *pcount, *pmembers;
  spl_SplObjectStorageElement *element;
  zend_long count;

  if (zend_parse_parameters(ZEND_NUM_ARGS(), "s", &buf, &buf_len) == FAILURE) {
    return;
  }

  if (buf_len == 0) {
    return;
  }

  /* storage */
  s = p = (const unsigned char*)buf;
  PHP_VAR_UNSERIALIZE_INIT(var_hash);

  if (*p!= 'x' || *++p != ':') {
    goto outexcept;
  }
  ++p;

  pcount = var_tmp_var(&var_hash);
  if (!php_var_unserialize(pcount, &p, s + buf_len, &var_hash) || Z_TYPE_P(pcount) != IS_LONG) {
    goto outexcept;
  }

  --p; /* for ';' */
  count = Z_LVAL_P(pcount);

  ZVAL_UNDEF(&entry);
  ZVAL_UNDEF(&inf);

  while (count-- > 0) {
    spl_SplObjectStorageElement *pelement;
    zend_hash_key key;

    if (*p != ';') {
      goto outexcept;
    }
    ++p;
    if(*p != 'O' && *p != 'C' && *p != 'r') {
      goto outexcept;
    }
    /* store reference to allow cross-references between different elements */
    if (!php_var_unserialize(&entry, &p, s + buf_len, &var_hash)) {
      goto outexcept;
    }
    if (*p == ',') { /* new version has inf */
      ++p;
      if (!php_var_unserialize(&inf, &p, s + buf_len, &var_hash)) {
        zval_ptr_dtor(&entry);
        goto outexcept;
      }
    }
    if (Z_TYPE(entry) != IS_OBJECT) {
      zval_ptr_dtor(&entry);
      zval_ptr_dtor(&inf);
      goto outexcept;
    }

    if (spl_object_storage_get_hash(&key, intern, getThis(), &entry) == FAILURE) {
      zval_ptr_dtor(&entry);
      zval_ptr_dtor(&inf);
      goto outexcept;
    }
    pelement = spl_object_storage_get(intern, &key);
    spl_object_storage_free_hash(intern, &key);
    if (pelement) {
      if (!Z_ISUNDEF(pelement->inf)) {
        var_push_dtor(&var_hash, &pelement->inf);
      }
      if (!Z_ISUNDEF(pelement->obj)) {
        var_push_dtor(&var_hash, &pelement->obj);
      }
    }
    element = spl_object_storage_attach(intern, getThis(), &entry, Z_ISUNDEF(inf)?NULL:&inf);
    var_replace(&var_hash, &entry, &element->obj);
    var_replace(&var_hash, &inf, &element->inf);
    zval_ptr_dtor(&entry);
    ZVAL_UNDEF(&entry);
    zval_ptr_dtor(&inf);
    ZVAL_UNDEF(&inf);
  }

  if (*p != ';') {
    goto outexcept;
  }
  ++p;

  /* members */
  if (*p!= 'm' || *++p != ':') {
    goto outexcept;
  }
  ++p;

  pmembers = var_tmp_var(&var_hash);
  if (!php_var_unserialize(pmembers, &p, s + buf_len, &var_hash) || Z_TYPE_P(pmembers) != IS_ARRAY) {
    goto outexcept;
  }

  /* copy members */
  object_properties_load(&intern->std, Z_ARRVAL_P(pmembers));

  PHP_VAR_UNSERIALIZE_DESTROY(var_hash);
  return;

outexcept:
  PHP_VAR_UNSERIALIZE_DESTROY(var_hash);
  zend_throw_exception_ex(spl_ce_UnexpectedValueException, 0, "Error at offset %zd of %zd bytes", ((char*)p - buf), buf_len);
  return;

} /* }}} */

ZEND_BEGIN_ARG_INFO(arginfo_Object, 0)
  ZEND_ARG_INFO(0, object)
ZEND_END_ARG_INFO();

ZEND_BEGIN_ARG_INFO_EX(arginfo_attach, 0, 0, 1)
  ZEND_ARG_INFO(0, object)
  ZEND_ARG_INFO(0, inf)
ZEND_END_ARG_INFO();

ZEND_BEGIN_ARG_INFO(arginfo_Serialized, 0)
  ZEND_ARG_INFO(0, serialized)
ZEND_END_ARG_INFO();

ZEND_BEGIN_ARG_INFO(arginfo_setInfo, 0)
  ZEND_ARG_INFO(0, info)
ZEND_END_ARG_INFO();

ZEND_BEGIN_ARG_INFO(arginfo_getHash, 0)
  ZEND_ARG_INFO(0, object)
ZEND_END_ARG_INFO();

ZEND_BEGIN_ARG_INFO_EX(arginfo_offsetGet, 0, 0, 1)
  ZEND_ARG_INFO(0, object)
ZEND_END_ARG_INFO()

ZEND_BEGIN_ARG_INFO(arginfo_splobject_void, 0)
ZEND_END_ARG_INFO()

static const zend_function_entry spl_funcs_SplObjectStorage[] = {
  SPL_ME(SplObjectStorage,  attach,      arginfo_attach,        0)
  SPL_ME(SplObjectStorage,  detach,      arginfo_Object,        0)
  SPL_ME(SplObjectStorage,  contains,    arginfo_Object,        0)
  SPL_ME(SplObjectStorage,  addAll,      arginfo_Object,        0)
  SPL_ME(SplObjectStorage,  removeAll,   arginfo_Object,        0)
  SPL_ME(SplObjectStorage,  removeAllExcept,   arginfo_Object,  0)
  SPL_ME(SplObjectStorage,  getInfo,     arginfo_splobject_void,0)
  SPL_ME(SplObjectStorage,  setInfo,     arginfo_setInfo,       0)
  SPL_ME(SplObjectStorage,  getHash,     arginfo_getHash,       0)
  /* Countable */
  SPL_ME(SplObjectStorage,  count,       arginfo_splobject_void,0)
  /* Iterator */
  SPL_ME(SplObjectStorage,  rewind,      arginfo_splobject_void,0)
  SPL_ME(SplObjectStorage,  valid,       arginfo_splobject_void,0)
  SPL_ME(SplObjectStorage,  key,         arginfo_splobject_void,0)
  SPL_ME(SplObjectStorage,  current,     arginfo_splobject_void,0)
  SPL_ME(SplObjectStorage,  next,        arginfo_splobject_void,0)
  /* Serializable */
  SPL_ME(SplObjectStorage,  unserialize, arginfo_Serialized,    0)
  SPL_ME(SplObjectStorage,  serialize,   arginfo_splobject_void,0)
  /* ArrayAccess */
  SPL_MA(SplObjectStorage, offsetExists, SplObjectStorage, contains, arginfo_offsetGet, 0)
  SPL_MA(SplObjectStorage, offsetSet,    SplObjectStorage, attach,   arginfo_attach, 0)
  SPL_MA(SplObjectStorage, offsetUnset,  SplObjectStorage, detach,   arginfo_offsetGet, 0)
  SPL_ME(SplObjectStorage, offsetGet,    arginfo_offsetGet,     0)
  PHP_FE_END
};

typedef enum {
  MIT_NEED_ANY     = 0,
  MIT_NEED_ALL     = 1,
  MIT_KEYS_NUMERIC = 0,
  MIT_KEYS_ASSOC   = 2
} MultipleIteratorFlags;

#define SPL_MULTIPLE_ITERATOR_GET_ALL_CURRENT   1
#define SPL_MULTIPLE_ITERATOR_GET_ALL_KEY       2

/* {{{ proto void MultipleIterator::__construct([int flags = MIT_NEED_ALL|MIT_KEYS_NUMERIC])
   Iterator that iterates over several iterators one after the other */
SPL_METHOD(MultipleIterator, __construct)
{
  spl_SplObjectStorage   *intern;
  zend_long               flags = MIT_NEED_ALL|MIT_KEYS_NUMERIC;

  if (zend_parse_parameters_throw(ZEND_NUM_ARGS(), "|l", &flags) == FAILURE) {
    return;
  }

  intern = Z_SPLOBJSTORAGE_P(getThis());
  intern->flags = flags;
}
/* }}} */

/* {{{ proto int MultipleIterator::getFlags()
   Return current flags */
SPL_METHOD(MultipleIterator, getFlags)
{
  spl_SplObjectStorage *intern = Z_SPLOBJSTORAGE_P(getThis());

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }
  RETURN_LONG(intern->flags);
}
/* }}} */

/* {{{ proto int MultipleIterator::setFlags(int flags)
   Set flags */
SPL_METHOD(MultipleIterator, setFlags)
{
  spl_SplObjectStorage *intern;
  intern = Z_SPLOBJSTORAGE_P(getThis());

  if (zend_parse_parameters(ZEND_NUM_ARGS(), "l", &intern->flags) == FAILURE) {
    return;
  }
}
/* }}} */

/* {{{ proto void attachIterator(Iterator iterator[, mixed info]) throws InvalidArgumentException
   Attach a new iterator */
SPL_METHOD(MultipleIterator, attachIterator)
{
  spl_SplObjectStorage        *intern;
  zval                        *iterator = NULL, *info = NULL;

  if (zend_parse_parameters(ZEND_NUM_ARGS(), "O|z!", &iterator, zend_ce_iterator, &info) == FAILURE) {
    return;
  }

  intern = Z_SPLOBJSTORAGE_P(getThis());

  if (info != NULL) {
    spl_SplObjectStorageElement *element;

    if (Z_TYPE_P(info) != IS_LONG && Z_TYPE_P(info) != IS_STRING) {
      zend_throw_exception(spl_ce_InvalidArgumentException, "Info must be NULL, integer or string", 0);
      return;
    }

    zend_hash_internal_pointer_reset_ex(&intern->storage, &intern->pos);
    while ((element = zend_hash_get_current_data_ptr_ex(&intern->storage, &intern->pos)) != NULL) {
      if (fast_is_identical_function(info, &element->inf)) {
        zend_throw_exception(spl_ce_InvalidArgumentException, "Key duplication error", 0);
        return;
      }
      zend_hash_move_forward_ex(&intern->storage, &intern->pos);
    }
  }

  spl_object_storage_attach(intern, getThis(), iterator, info);
}
/* }}} */

/* {{{ proto void MultipleIterator::rewind()
   Rewind all attached iterator instances */
SPL_METHOD(MultipleIterator, rewind)
{
  spl_SplObjectStorage        *intern;
  spl_SplObjectStorageElement *element;
  zval                        *it;

  intern = Z_SPLOBJSTORAGE_P(getThis());

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  zend_hash_internal_pointer_reset_ex(&intern->storage, &intern->pos);
  while ((element = zend_hash_get_current_data_ptr_ex(&intern->storage, &intern->pos)) != NULL && !EG(exception)) {
    it = &element->obj;
    zend_call_method_with_0_params(it, Z_OBJCE_P(it), &Z_OBJCE_P(it)->iterator_funcs.zf_rewind, "rewind", NULL);
    zend_hash_move_forward_ex(&intern->storage, &intern->pos);
  }
}
/* }}} */

/* {{{ proto void MultipleIterator::next()
   Move all attached iterator instances forward */
SPL_METHOD(MultipleIterator, next)
{
  spl_SplObjectStorage        *intern;
  spl_SplObjectStorageElement *element;
  zval                        *it;

  intern = Z_SPLOBJSTORAGE_P(getThis());

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  zend_hash_internal_pointer_reset_ex(&intern->storage, &intern->pos);
  while ((element = zend_hash_get_current_data_ptr_ex(&intern->storage, &intern->pos)) != NULL && !EG(exception)) {
    it = &element->obj;
    zend_call_method_with_0_params(it, Z_OBJCE_P(it), &Z_OBJCE_P(it)->iterator_funcs.zf_next, "next", NULL);
    zend_hash_move_forward_ex(&intern->storage, &intern->pos);
  }
}
/* }}} */

/* {{{ proto bool MultipleIterator::valid()
   Return whether all or one sub iterator is valid depending on flags */
SPL_METHOD(MultipleIterator, valid)
{
  spl_SplObjectStorage        *intern;
  spl_SplObjectStorageElement *element;
  zval                        *it, retval;
  zend_long                         expect, valid;

  intern = Z_SPLOBJSTORAGE_P(getThis());

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  if (!zend_hash_num_elements(&intern->storage)) {
    RETURN_FALSE;
  }

  expect = (intern->flags & MIT_NEED_ALL) ? 1 : 0;

  zend_hash_internal_pointer_reset_ex(&intern->storage, &intern->pos);
  while ((element = zend_hash_get_current_data_ptr_ex(&intern->storage, &intern->pos)) != NULL && !EG(exception)) {
    it = &element->obj;
    zend_call_method_with_0_params(it, Z_OBJCE_P(it), &Z_OBJCE_P(it)->iterator_funcs.zf_valid, "valid", &retval);

    if (!Z_ISUNDEF(retval)) {
      valid = (Z_TYPE(retval) == IS_TRUE);
      zval_ptr_dtor(&retval);
    } else {
      valid = 0;
    }

    if (expect != valid) {
      RETURN_BOOL(!expect);
    }

    zend_hash_move_forward_ex(&intern->storage, &intern->pos);
  }

  RETURN_BOOL(expect);
}
/* }}} */

static void spl_multiple_iterator_get_all(spl_SplObjectStorage *intern, int get_type, zval *return_value) /* {{{ */
{
  spl_SplObjectStorageElement *element;
  zval                        *it, retval;
  int                          valid = 1, num_elements;

  num_elements = zend_hash_num_elements(&intern->storage);
  if (num_elements < 1) {
    RETURN_FALSE;
  }

  array_init_size(return_value, num_elements);

  zend_hash_internal_pointer_reset_ex(&intern->storage, &intern->pos);
  while ((element = zend_hash_get_current_data_ptr_ex(&intern->storage, &intern->pos)) != NULL && !EG(exception)) {
    it = &element->obj;
    zend_call_method_with_0_params(it, Z_OBJCE_P(it), &Z_OBJCE_P(it)->iterator_funcs.zf_valid, "valid", &retval);

    if (!Z_ISUNDEF(retval)) {
      valid = Z_TYPE(retval) == IS_TRUE;
      zval_ptr_dtor(&retval);
    } else {
      valid = 0;
    }

    if (valid) {
      if (SPL_MULTIPLE_ITERATOR_GET_ALL_CURRENT == get_type) {
        zend_call_method_with_0_params(it, Z_OBJCE_P(it), &Z_OBJCE_P(it)->iterator_funcs.zf_current, "current", &retval);
      } else {
        zend_call_method_with_0_params(it, Z_OBJCE_P(it), &Z_OBJCE_P(it)->iterator_funcs.zf_key, "key", &retval);
      }
      if (Z_ISUNDEF(retval)) {
        zend_throw_exception(spl_ce_RuntimeException, "Failed to call sub iterator method", 0);
        return;
      }
    } else if (intern->flags & MIT_NEED_ALL) {
      if (SPL_MULTIPLE_ITERATOR_GET_ALL_CURRENT == get_type) {
        zend_throw_exception(spl_ce_RuntimeException, "Called current() with non valid sub iterator", 0);
      } else {
        zend_throw_exception(spl_ce_RuntimeException, "Called key() with non valid sub iterator", 0);
      }
      return;
    } else {
      ZVAL_NULL(&retval);
    }

    if (intern->flags & MIT_KEYS_ASSOC) {
      switch (Z_TYPE(element->inf)) {
        case IS_LONG:
          add_index_zval(return_value, Z_LVAL(element->inf), &retval);
          break;
        case IS_STRING:
          zend_symtable_update(Z_ARRVAL_P(return_value), Z_STR(element->inf), &retval);
          break;
        default:
          zval_ptr_dtor(&retval);
          zend_throw_exception(spl_ce_InvalidArgumentException, "Sub-Iterator is associated with NULL", 0);
          return;
      }
    } else {
      add_next_index_zval(return_value, &retval);
    }

    zend_hash_move_forward_ex(&intern->storage, &intern->pos);
  }
}
/* }}} */

/* {{{ proto array current() throws RuntimeException throws InvalidArgumentException
   Return an array of all registered Iterator instances current() result */
SPL_METHOD(MultipleIterator, current)
{
  spl_SplObjectStorage        *intern;
  intern = Z_SPLOBJSTORAGE_P(getThis());

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  spl_multiple_iterator_get_all(intern, SPL_MULTIPLE_ITERATOR_GET_ALL_CURRENT, return_value);
}
/* }}} */

/* {{{ proto array MultipleIterator::key()
   Return an array of all registered Iterator instances key() result */
SPL_METHOD(MultipleIterator, key)
{
  spl_SplObjectStorage *intern;
  intern = Z_SPLOBJSTORAGE_P(getThis());

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  spl_multiple_iterator_get_all(intern, SPL_MULTIPLE_ITERATOR_GET_ALL_KEY, return_value);
}
/* }}} */

ZEND_BEGIN_ARG_INFO_EX(arginfo_MultipleIterator_attachIterator, 0, 0, 1)
  ZEND_ARG_OBJ_INFO(0, iterator, Iterator, 0)
  ZEND_ARG_INFO(0, infos)
ZEND_END_ARG_INFO();

ZEND_BEGIN_ARG_INFO_EX(arginfo_MultipleIterator_detachIterator, 0, 0, 1)
  ZEND_ARG_OBJ_INFO(0, iterator, Iterator, 0)
ZEND_END_ARG_INFO();

ZEND_BEGIN_ARG_INFO_EX(arginfo_MultipleIterator_containsIterator, 0, 0, 1)
  ZEND_ARG_OBJ_INFO(0, iterator, Iterator, 0)
ZEND_END_ARG_INFO();

ZEND_BEGIN_ARG_INFO_EX(arginfo_MultipleIterator_setflags, 0, 0, 1)
  ZEND_ARG_INFO(0, flags)
ZEND_END_ARG_INFO();

static const zend_function_entry spl_funcs_MultipleIterator[] = {
  SPL_ME(MultipleIterator,  __construct,            arginfo_MultipleIterator_setflags,          0)
  SPL_ME(MultipleIterator,  getFlags,               arginfo_splobject_void,                     0)
  SPL_ME(MultipleIterator,  setFlags,               arginfo_MultipleIterator_setflags,          0)
  SPL_ME(MultipleIterator,  attachIterator,         arginfo_MultipleIterator_attachIterator,    0)
  SPL_MA(MultipleIterator,  detachIterator,         SplObjectStorage, detach,   arginfo_MultipleIterator_detachIterator,   0)
  SPL_MA(MultipleIterator,  containsIterator,       SplObjectStorage, contains, arginfo_MultipleIterator_containsIterator, 0)
  SPL_MA(MultipleIterator,  countIterators,         SplObjectStorage, count,    arginfo_splobject_void,                    0)
  /* Iterator */
  SPL_ME(MultipleIterator,  rewind,                 arginfo_splobject_void,                     0)
  SPL_ME(MultipleIterator,  valid,                  arginfo_splobject_void,                     0)
  SPL_ME(MultipleIterator,  key,                    arginfo_splobject_void,                     0)
  SPL_ME(MultipleIterator,  current,                arginfo_splobject_void,                     0)
  SPL_ME(MultipleIterator,  next,                   arginfo_splobject_void,                     0)
  PHP_FE_END
};

/* {{{ PHP_MINIT_FUNCTION(spl_observer) */
PHP_MINIT_FUNCTION(spl_observer)
{
  REGISTER_SPL_INTERFACE(SplObserver);
  REGISTER_SPL_INTERFACE(SplSubject);

  REGISTER_SPL_STD_CLASS_EX(SplObjectStorage, spl_SplObjectStorage_new, spl_funcs_SplObjectStorage);
  memcpy(&spl_handler_SplObjectStorage, zend_get_std_object_handlers(), sizeof(zend_object_handlers));

  spl_handler_SplObjectStorage.offset          = XtOffsetOf(spl_SplObjectStorage, std);
  spl_handler_SplObjectStorage.get_debug_info  = spl_object_storage_debug_info;
  spl_handler_SplObjectStorage.compare_objects = spl_object_storage_compare_objects;
  spl_handler_SplObjectStorage.clone_obj       = spl_object_storage_clone;
  spl_handler_SplObjectStorage.get_gc          = spl_object_storage_get_gc;
  spl_handler_SplObjectStorage.dtor_obj        = zend_objects_destroy_object;
  spl_handler_SplObjectStorage.free_obj        = spl_SplObjectStorage_free_storage;

  REGISTER_SPL_IMPLEMENTS(SplObjectStorage, Countable);
  REGISTER_SPL_IMPLEMENTS(SplObjectStorage, Iterator);
  REGISTER_SPL_IMPLEMENTS(SplObjectStorage, Serializable);
  REGISTER_SPL_IMPLEMENTS(SplObjectStorage, ArrayAccess);

  REGISTER_SPL_STD_CLASS_EX(MultipleIterator, spl_SplObjectStorage_new, spl_funcs_MultipleIterator);
  REGISTER_SPL_ITERATOR(MultipleIterator);

  REGISTER_SPL_CLASS_CONST_LONG(MultipleIterator, "MIT_NEED_ANY",     MIT_NEED_ANY);
  REGISTER_SPL_CLASS_CONST_LONG(MultipleIterator, "MIT_NEED_ALL",     MIT_NEED_ALL);
  REGISTER_SPL_CLASS_CONST_LONG(MultipleIterator, "MIT_KEYS_NUMERIC", MIT_KEYS_NUMERIC);
  REGISTER_SPL_CLASS_CONST_LONG(MultipleIterator, "MIT_KEYS_ASSOC",   MIT_KEYS_ASSOC);

  return SUCCESS;
}
/* }}} */

/*
 * Local variables:
 * tab-width: 4
 * c-basic-offset: 4
 * End:
 * vim600: fdm=marker
 * vim: noet sw=4 ts=4
 */
```

```c
/*
   +----------------------------------------------------------------------+
   | PHP Version 7                                                        |
   +----------------------------------------------------------------------+
   | Copyright (c) 1997-2018 The PHP Group                                |
   +----------------------------------------------------------------------+
   | This source file is subject to version 3.01 of the PHP license,      |
   | that is bundled with this package in the file LICENSE, and is        |
   | available through the world-wide-web at the following url:           |
   | http://www.php.net/license/3_01.txt                                  |
   | If you did not receive a copy of the PHP license and are unable to   |
   | obtain it through the world-wide-web, please send a note to          |
   | license@php.net so we can mail you a copy immediately.               |
   +----------------------------------------------------------------------+
   | Authors: Etienne Kneuss <colder@php.net>                             |
   +----------------------------------------------------------------------+
 */

/* $Id$ */

#ifdef HAVE_CONFIG_H
# include "config.h"
#endif

#include "php.h"
#include "zend_exceptions.h"
#include "zend_hash.h"

#include "php_spl.h"
#include "ext/standard/info.h"
#include "ext/standard/php_var.h"
#include "zend_smart_str.h"
#include "spl_functions.h"
#include "spl_engine.h"
#include "spl_iterators.h"
#include "spl_dllist.h"
#include "spl_exceptions.h"

zend_object_handlers spl_handler_SplDoublyLinkedList;
PHPAPI zend_class_entry  *spl_ce_SplDoublyLinkedList;
PHPAPI zend_class_entry  *spl_ce_SplQueue;
PHPAPI zend_class_entry  *spl_ce_SplStack;

#define SPL_LLIST_DELREF(elem) if(!--(elem)->rc) { \
  efree(elem); \
}

#define SPL_LLIST_CHECK_DELREF(elem) if((elem) && !--(elem)->rc) { \
  efree(elem); \
}

#define SPL_LLIST_ADDREF(elem) (elem)->rc++
#define SPL_LLIST_CHECK_ADDREF(elem) if(elem) (elem)->rc++

#define SPL_DLLIST_IT_DELETE 0x00000001 /* Delete flag makes the iterator delete the current element on next */
#define SPL_DLLIST_IT_LIFO   0x00000002 /* LIFO flag makes the iterator traverse the structure as a LastInFirstOut */
#define SPL_DLLIST_IT_MASK   0x00000003 /* Mask to isolate flags related to iterators */
#define SPL_DLLIST_IT_FIX    0x00000004 /* Backward/Forward bit is fixed */

#ifdef accept
#undef accept
#endif

typedef struct _spl_ptr_llist_element {
  struct _spl_ptr_llist_element *prev;
  struct _spl_ptr_llist_element *next;
  int                            rc;
  zval                           data;
} spl_ptr_llist_element;

typedef void (*spl_ptr_llist_dtor_func)(spl_ptr_llist_element *);
typedef void (*spl_ptr_llist_ctor_func)(spl_ptr_llist_element *);

typedef struct _spl_ptr_llist {
  spl_ptr_llist_element   *head;
  spl_ptr_llist_element   *tail;
  spl_ptr_llist_dtor_func  dtor;
  spl_ptr_llist_ctor_func  ctor;
  int count;
} spl_ptr_llist;

typedef struct _spl_dllist_object spl_dllist_object;
typedef struct _spl_dllist_it spl_dllist_it;

struct _spl_dllist_object {
  spl_ptr_llist         *llist;
  int                    traverse_position;
  spl_ptr_llist_element *traverse_pointer;
  int                    flags;
  zend_function         *fptr_offset_get;
  zend_function         *fptr_offset_set;
  zend_function         *fptr_offset_has;
  zend_function         *fptr_offset_del;
  zend_function         *fptr_count;
  zend_class_entry      *ce_get_iterator;
  zval                  *gc_data;
  int                    gc_data_count;
  zend_object            std;
};

/* define an overloaded iterator structure */
struct _spl_dllist_it {
  zend_user_iterator     intern;
  spl_ptr_llist_element *traverse_pointer;
  int                    traverse_position;
  int                    flags;
};

static inline spl_dllist_object *spl_dllist_from_obj(zend_object *obj) /* {{{ */ {
  return (spl_dllist_object*)((char*)(obj) - XtOffsetOf(spl_dllist_object, std));
}
/* }}} */

#define Z_SPLDLLIST_P(zv)  spl_dllist_from_obj(Z_OBJ_P((zv)))

/* {{{  spl_ptr_llist */
static void spl_ptr_llist_zval_dtor(spl_ptr_llist_element *elem) { /* {{{ */
  if (!Z_ISUNDEF(elem->data)) {
    zval_ptr_dtor(&elem->data);
    ZVAL_UNDEF(&elem->data);
  }
}
/* }}} */

static void spl_ptr_llist_zval_ctor(spl_ptr_llist_element *elem) { /* {{{ */
  if (Z_REFCOUNTED(elem->data)) {
    Z_ADDREF(elem->data);
  }
}
/* }}} */

static spl_ptr_llist *spl_ptr_llist_init(spl_ptr_llist_ctor_func ctor, spl_ptr_llist_dtor_func dtor) /* {{{ */
{
  spl_ptr_llist *llist = emalloc(sizeof(spl_ptr_llist));

  llist->head  = NULL;
  llist->tail  = NULL;
  llist->count = 0;
  llist->dtor  = dtor;
  llist->ctor  = ctor;

  return llist;
}
/* }}} */

static zend_long spl_ptr_llist_count(spl_ptr_llist *llist) /* {{{ */
{
  return (zend_long)llist->count;
}
/* }}} */

static void spl_ptr_llist_destroy(spl_ptr_llist *llist) /* {{{ */
{
  spl_ptr_llist_element   *current = llist->head, *next;
  spl_ptr_llist_dtor_func  dtor    = llist->dtor;

  while (current) {
    next = current->next;
    if (dtor) {
      dtor(current);
    }
    SPL_LLIST_DELREF(current);
    current = next;
  }

  efree(llist);
}
/* }}} */

static spl_ptr_llist_element *spl_ptr_llist_offset(spl_ptr_llist *llist, zend_long offset, int backward) /* {{{ */
{

  spl_ptr_llist_element *current;
  int pos = 0;

  if (backward) {
    current = llist->tail;
  } else {
    current = llist->head;
  }

  while (current && pos < offset) {
    pos++;
    if (backward) {
      current = current->prev;
    } else {
      current = current->next;
    }
  }

  return current;
}
/* }}} */

static void spl_ptr_llist_unshift(spl_ptr_llist *llist, zval *data) /* {{{ */
{
  spl_ptr_llist_element *elem = emalloc(sizeof(spl_ptr_llist_element));

  elem->rc   = 1;
  elem->prev = NULL;
  elem->next = llist->head;
  ZVAL_COPY_VALUE(&elem->data, data);

  if (llist->head) {
    llist->head->prev = elem;
  } else {
    llist->tail = elem;
  }

  llist->head = elem;
  llist->count++;

  if (llist->ctor) {
    llist->ctor(elem);
  }
}
/* }}} */

static void spl_ptr_llist_push(spl_ptr_llist *llist, zval *data) /* {{{ */
{
  spl_ptr_llist_element *elem = emalloc(sizeof(spl_ptr_llist_element));

  elem->rc   = 1;
  elem->prev = llist->tail;
  elem->next = NULL;
  ZVAL_COPY_VALUE(&elem->data, data);

  if (llist->tail) {
    llist->tail->next = elem;
  } else {
    llist->head = elem;
  }

  llist->tail = elem;
  llist->count++;

  if (llist->ctor) {
    llist->ctor(elem);
  }
}
/* }}} */

static void spl_ptr_llist_pop(spl_ptr_llist *llist, zval *ret) /* {{{ */
{
  spl_ptr_llist_element    *tail = llist->tail;

  if (tail == NULL) {
    ZVAL_UNDEF(ret);
    return;
  }

  if (tail->prev) {
    tail->prev->next = NULL;
  } else {
    llist->head = NULL;
  }

  llist->tail = tail->prev;
  llist->count--;
  ZVAL_COPY(ret, &tail->data);

  if (llist->dtor) {
    llist->dtor(tail);
  }

  ZVAL_UNDEF(&tail->data);

  SPL_LLIST_DELREF(tail);
}
/* }}} */

static zval *spl_ptr_llist_last(spl_ptr_llist *llist) /* {{{ */
{
  spl_ptr_llist_element *tail = llist->tail;

  if (tail == NULL) {
    return NULL;
  } else {
    return &tail->data;
  }
}
/* }}} */

static zval *spl_ptr_llist_first(spl_ptr_llist *llist) /* {{{ */
{
  spl_ptr_llist_element *head = llist->head;

  if (head == NULL) {
    return NULL;
  } else {
    return &head->data;
  }
}
/* }}} */

static void spl_ptr_llist_shift(spl_ptr_llist *llist, zval *ret) /* {{{ */
{
  spl_ptr_llist_element   *head = llist->head;

  if (head == NULL) {
    ZVAL_UNDEF(ret);
    return;
  }

  if (head->next) {
    head->next->prev = NULL;
  } else {
    llist->tail = NULL;
  }

  llist->head = head->next;
  llist->count--;
  ZVAL_COPY(ret, &head->data);

  if (llist->dtor) {
    llist->dtor(head);
  }
  ZVAL_UNDEF(&head->data);

  SPL_LLIST_DELREF(head);
}
/* }}} */

static void spl_ptr_llist_copy(spl_ptr_llist *from, spl_ptr_llist *to) /* {{{ */
{
  spl_ptr_llist_element *current = from->head, *next;
//??? spl_ptr_llist_ctor_func ctor = from->ctor;

  while (current) {
    next = current->next;

    /*??? FIXME
    if (ctor) {
      ctor(current);
    }
    */

    spl_ptr_llist_push(to, &current->data);
    current = next;
  }

}
/* }}} */

/* }}} */

static void spl_dllist_object_free_storage(zend_object *object) /* {{{ */
{
  spl_dllist_object *intern = spl_dllist_from_obj(object);
  zval tmp;

  zend_object_std_dtor(&intern->std);

  while (intern->llist->count > 0) {
    spl_ptr_llist_pop(intern->llist, &tmp);
    zval_ptr_dtor(&tmp);
  }

  if (intern->gc_data != NULL) {
    efree(intern->gc_data);
  };

  spl_ptr_llist_destroy(intern->llist);
  SPL_LLIST_CHECK_DELREF(intern->traverse_pointer);
}
/* }}} */

zend_object_iterator *spl_dllist_get_iterator(zend_class_entry *ce, zval *object, int by_ref);

static zend_object *spl_dllist_object_new_ex(zend_class_entry *class_type, zval *orig, int clone_orig) /* {{{ */
{
  spl_dllist_object *intern;
  zend_class_entry  *parent = class_type;
  int                inherited = 0;

  intern = zend_object_alloc(sizeof(spl_dllist_object), parent);

  zend_object_std_init(&intern->std, class_type);
  object_properties_init(&intern->std, class_type);

  intern->flags = 0;
  intern->traverse_position = 0;

  if (orig) {
    spl_dllist_object *other = Z_SPLDLLIST_P(orig);
    intern->ce_get_iterator = other->ce_get_iterator;

    if (clone_orig) {
      intern->llist = (spl_ptr_llist *)spl_ptr_llist_init(other->llist->ctor, other->llist->dtor);
      spl_ptr_llist_copy(other->llist, intern->llist);
      intern->traverse_pointer  = intern->llist->head;
      SPL_LLIST_CHECK_ADDREF(intern->traverse_pointer);
    } else {
      intern->llist = other->llist;
      intern->traverse_pointer  = intern->llist->head;
      SPL_LLIST_CHECK_ADDREF(intern->traverse_pointer);
    }

    intern->flags = other->flags;
  } else {
    intern->llist = (spl_ptr_llist *)spl_ptr_llist_init(spl_ptr_llist_zval_ctor, spl_ptr_llist_zval_dtor);
    intern->traverse_pointer  = intern->llist->head;
    SPL_LLIST_CHECK_ADDREF(intern->traverse_pointer);
  }

  while (parent) {
    if (parent == spl_ce_SplStack) {
      intern->flags |= (SPL_DLLIST_IT_FIX | SPL_DLLIST_IT_LIFO);
      intern->std.handlers = &spl_handler_SplDoublyLinkedList;
    } else if (parent == spl_ce_SplQueue) {
      intern->flags |= SPL_DLLIST_IT_FIX;
      intern->std.handlers = &spl_handler_SplDoublyLinkedList;
    }

    if (parent == spl_ce_SplDoublyLinkedList) {
      intern->std.handlers = &spl_handler_SplDoublyLinkedList;
      break;
    }

    parent = parent->parent;
    inherited = 1;
  }

  if (!parent) { /* this must never happen */
    php_error_docref(NULL, E_COMPILE_ERROR, "Internal compiler error, Class is not child of SplDoublyLinkedList");
  }
  if (inherited) {
    intern->fptr_offset_get = zend_hash_str_find_ptr(&class_type->function_table, "offsetget", sizeof("offsetget") - 1);
    if (intern->fptr_offset_get->common.scope == parent) {
      intern->fptr_offset_get = NULL;
    }
    intern->fptr_offset_set = zend_hash_str_find_ptr(&class_type->function_table, "offsetset", sizeof("offsetset") - 1);
    if (intern->fptr_offset_set->common.scope == parent) {
      intern->fptr_offset_set = NULL;
    }
    intern->fptr_offset_has = zend_hash_str_find_ptr(&class_type->function_table, "offsetexists", sizeof("offsetexists") - 1);
    if (intern->fptr_offset_has->common.scope == parent) {
      intern->fptr_offset_has = NULL;
    }
    intern->fptr_offset_del = zend_hash_str_find_ptr(&class_type->function_table, "offsetunset", sizeof("offsetunset") - 1);
    if (intern->fptr_offset_del->common.scope == parent) {
      intern->fptr_offset_del = NULL;
    }
    intern->fptr_count = zend_hash_str_find_ptr(&class_type->function_table, "count", sizeof("count") - 1);
    if (intern->fptr_count->common.scope == parent) {
      intern->fptr_count = NULL;
    }
  }

  return &intern->std;
}
/* }}} */

static zend_object *spl_dllist_object_new(zend_class_entry *class_type) /* {{{ */
{
  return spl_dllist_object_new_ex(class_type, NULL, 0);
}
/* }}} */

static zend_object *spl_dllist_object_clone(zval *zobject) /* {{{ */
{
  zend_object        *old_object;
  zend_object        *new_object;

  old_object  = Z_OBJ_P(zobject);
  new_object = spl_dllist_object_new_ex(old_object->ce, zobject, 1);

  zend_objects_clone_members(new_object, old_object);

  return new_object;
}
/* }}} */

static int spl_dllist_object_count_elements(zval *object, zend_long *count) /* {{{ */
{
  spl_dllist_object *intern = Z_SPLDLLIST_P(object);

  if (intern->fptr_count) {
    zval rv;
    zend_call_method_with_0_params(object, intern->std.ce, &intern->fptr_count, "count", &rv);
    if (!Z_ISUNDEF(rv)) {
      *count = zval_get_long(&rv);
      zval_ptr_dtor(&rv);
      return SUCCESS;
    }
    *count = 0;
    return FAILURE;
  }

  *count = spl_ptr_llist_count(intern->llist);
  return SUCCESS;
}
/* }}} */

static HashTable* spl_dllist_object_get_debug_info(zval *obj, int *is_temp) /* {{{{ */
{
  spl_dllist_object     *intern  = Z_SPLDLLIST_P(obj);
  spl_ptr_llist_element *current = intern->llist->head, *next;
  zval tmp, dllist_array;
  zend_string *pnstr;
  int  i = 0;
  HashTable *debug_info;
  *is_temp = 1;

  if (!intern->std.properties) {
    rebuild_object_properties(&intern->std);
  }

  debug_info = zend_new_array(1);
  zend_hash_copy(debug_info, intern->std.properties, (copy_ctor_func_t) zval_add_ref);

  pnstr = spl_gen_private_prop_name(spl_ce_SplDoublyLinkedList, "flags", sizeof("flags")-1);
  ZVAL_LONG(&tmp, intern->flags);
  zend_hash_add(debug_info, pnstr, &tmp);
  zend_string_release(pnstr);

  array_init(&dllist_array);

  while (current) {
    next = current->next;

    add_index_zval(&dllist_array, i, &current->data);
    if (Z_REFCOUNTED(current->data)) {
      Z_ADDREF(current->data);
    }
    i++;

    current = next;
  }

  pnstr = spl_gen_private_prop_name(spl_ce_SplDoublyLinkedList, "dllist", sizeof("dllist")-1);
  zend_hash_add(debug_info, pnstr, &dllist_array);
  zend_string_release(pnstr);

  return debug_info;
}
/* }}}} */

static HashTable *spl_dllist_object_get_gc(zval *obj, zval **gc_data, int *gc_data_count) /* {{{ */
{
  spl_dllist_object *intern  = Z_SPLDLLIST_P(obj);
  spl_ptr_llist_element *current = intern->llist->head;
  int i = 0;

  if (intern->gc_data_count < intern->llist->count) {
    intern->gc_data_count = intern->llist->count;
    intern->gc_data = safe_erealloc(intern->gc_data, intern->gc_data_count, sizeof(zval), 0);
  }

  while (current) {
    ZVAL_COPY_VALUE(&intern->gc_data[i++], &current->data);
    current = current->next;
  }

  *gc_data = intern->gc_data;
  *gc_data_count = i;
  return zend_std_get_properties(obj);
}
/* }}} */

/* {{{ proto bool SplDoublyLinkedList::push(mixed value)
     Push $value on the SplDoublyLinkedList */
SPL_METHOD(SplDoublyLinkedList, push)
{
  zval *value;
  spl_dllist_object *intern;

  if (zend_parse_parameters(ZEND_NUM_ARGS(), "z", &value) == FAILURE) {
    return;
  }

  intern = Z_SPLDLLIST_P(getThis());
  spl_ptr_llist_push(intern->llist, value);

  RETURN_TRUE;
}
/* }}} */

/* {{{ proto bool SplDoublyLinkedList::unshift(mixed value)
     Unshift $value on the SplDoublyLinkedList */
SPL_METHOD(SplDoublyLinkedList, unshift)
{
  zval *value;
  spl_dllist_object *intern;

  if (zend_parse_parameters(ZEND_NUM_ARGS(), "z", &value) == FAILURE) {
    return;
  }

  intern = Z_SPLDLLIST_P(getThis());
  spl_ptr_llist_unshift(intern->llist, value);

  RETURN_TRUE;
}
/* }}} */

/* {{{ proto mixed SplDoublyLinkedList::pop()
     Pop an element out of the SplDoublyLinkedList */
SPL_METHOD(SplDoublyLinkedList, pop)
{
  spl_dllist_object *intern;

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  intern = Z_SPLDLLIST_P(getThis());
  spl_ptr_llist_pop(intern->llist, return_value);

  if (Z_ISUNDEF_P(return_value)) {
    zend_throw_exception(spl_ce_RuntimeException, "Can't pop from an empty datastructure", 0);
    RETURN_NULL();
  }
}
/* }}} */

/* {{{ proto mixed SplDoublyLinkedList::shift()
     Shift an element out of the SplDoublyLinkedList */
SPL_METHOD(SplDoublyLinkedList, shift)
{
  spl_dllist_object *intern;

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  intern = Z_SPLDLLIST_P(getThis());
  spl_ptr_llist_shift(intern->llist, return_value);

  if (Z_ISUNDEF_P(return_value)) {
    zend_throw_exception(spl_ce_RuntimeException, "Can't shift from an empty datastructure", 0);
    RETURN_NULL();
  }
}
/* }}} */

/* {{{ proto mixed SplDoublyLinkedList::top()
     Peek at the top element of the SplDoublyLinkedList */
SPL_METHOD(SplDoublyLinkedList, top)
{
  zval *value;
  spl_dllist_object *intern;

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  intern = Z_SPLDLLIST_P(getThis());
  value = spl_ptr_llist_last(intern->llist);

  if (value == NULL || Z_ISUNDEF_P(value)) {
    zend_throw_exception(spl_ce_RuntimeException, "Can't peek at an empty datastructure", 0);
    return;
  }

  ZVAL_DEREF(value);
  ZVAL_COPY(return_value, value);
}
/* }}} */

/* {{{ proto mixed SplDoublyLinkedList::bottom()
     Peek at the bottom element of the SplDoublyLinkedList */
SPL_METHOD(SplDoublyLinkedList, bottom)
{
  zval *value;
  spl_dllist_object *intern;

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  intern = Z_SPLDLLIST_P(getThis());
  value  = spl_ptr_llist_first(intern->llist);

  if (value == NULL || Z_ISUNDEF_P(value)) {
    zend_throw_exception(spl_ce_RuntimeException, "Can't peek at an empty datastructure", 0);
    return;
  }

  ZVAL_DEREF(value);
  ZVAL_COPY(return_value, value);
}
/* }}} */

/* {{{ proto int SplDoublyLinkedList::count()
 Return the number of elements in the datastructure. */
SPL_METHOD(SplDoublyLinkedList, count)
{
  zend_long count;
  spl_dllist_object *intern = Z_SPLDLLIST_P(getThis());

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  count = spl_ptr_llist_count(intern->llist);
  RETURN_LONG(count);
}
/* }}} */

/* {{{ proto int SplDoublyLinkedList::isEmpty()
 Return true if the SplDoublyLinkedList is empty. */
SPL_METHOD(SplDoublyLinkedList, isEmpty)
{
  zend_long count;

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  spl_dllist_object_count_elements(getThis(), &count);
  RETURN_BOOL(count == 0);
}
/* }}} */

/* {{{ proto int SplDoublyLinkedList::setIteratorMode(int flags)
 Set the mode of iteration */
SPL_METHOD(SplDoublyLinkedList, setIteratorMode)
{
  zend_long value;
  spl_dllist_object *intern;

  if (zend_parse_parameters(ZEND_NUM_ARGS(), "l", &value) == FAILURE) {
    return;
  }

  intern = Z_SPLDLLIST_P(getThis());

  if (intern->flags & SPL_DLLIST_IT_FIX
    && (intern->flags & SPL_DLLIST_IT_LIFO) != (value & SPL_DLLIST_IT_LIFO)) {
    zend_throw_exception(spl_ce_RuntimeException, "Iterators' LIFO/FIFO modes for SplStack/SplQueue objects are frozen", 0);
    return;
  }

  intern->flags = (value & SPL_DLLIST_IT_MASK) | (intern->flags & SPL_DLLIST_IT_FIX);

  RETURN_LONG(intern->flags);
}
/* }}} */

/* {{{ proto int SplDoublyLinkedList::getIteratorMode()
 Return the mode of iteration */
SPL_METHOD(SplDoublyLinkedList, getIteratorMode)
{
  spl_dllist_object *intern;

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  intern = Z_SPLDLLIST_P(getThis());

  RETURN_LONG(intern->flags);
}
/* }}} */

/* {{{ proto bool SplDoublyLinkedList::offsetExists(mixed index)
 Returns whether the requested $index exists. */
SPL_METHOD(SplDoublyLinkedList, offsetExists)
{
  zval              *zindex;
  spl_dllist_object *intern;
  zend_long               index;

  if (zend_parse_parameters(ZEND_NUM_ARGS(), "z", &zindex) == FAILURE) {
    return;
  }

  intern = Z_SPLDLLIST_P(getThis());
  index  = spl_offset_convert_to_long(zindex);

  RETURN_BOOL(index >= 0 && index < intern->llist->count);
} /* }}} */

/* {{{ proto mixed SplDoublyLinkedList::offsetGet(mixed index)
 Returns the value at the specified $index. */
SPL_METHOD(SplDoublyLinkedList, offsetGet)
{
  zval                  *zindex;
  zend_long                   index;
  spl_dllist_object     *intern;
  spl_ptr_llist_element *element;

  if (zend_parse_parameters(ZEND_NUM_ARGS(), "z", &zindex) == FAILURE) {
    return;
  }

  intern = Z_SPLDLLIST_P(getThis());
  index  = spl_offset_convert_to_long(zindex);

  if (index < 0 || index >= intern->llist->count) {
    zend_throw_exception(spl_ce_OutOfRangeException, "Offset invalid or out of range", 0);
    return;
  }

  element = spl_ptr_llist_offset(intern->llist, index, intern->flags & SPL_DLLIST_IT_LIFO);

  if (element != NULL) {
    zval *value = &element->data;

    ZVAL_DEREF(value);
    ZVAL_COPY(return_value, value);
  } else {
    zend_throw_exception(spl_ce_OutOfRangeException, "Offset invalid", 0);
  }
} /* }}} */

/* {{{ proto void SplDoublyLinkedList::offsetSet(mixed index, mixed newval)
 Sets the value at the specified $index to $newval. */
SPL_METHOD(SplDoublyLinkedList, offsetSet)
{
  zval                  *zindex, *value;
  spl_dllist_object     *intern;

  if (zend_parse_parameters(ZEND_NUM_ARGS(), "zz", &zindex, &value) == FAILURE) {
    return;
  }

  intern = Z_SPLDLLIST_P(getThis());

  if (Z_TYPE_P(zindex) == IS_NULL) {
    /* $obj[] = ... */
    spl_ptr_llist_push(intern->llist, value);
  } else {
    /* $obj[$foo] = ... */
    zend_long                   index;
    spl_ptr_llist_element *element;

    index = spl_offset_convert_to_long(zindex);

    if (index < 0 || index >= intern->llist->count) {
      zend_throw_exception(spl_ce_OutOfRangeException, "Offset invalid or out of range", 0);
      return;
    }

    element = spl_ptr_llist_offset(intern->llist, index, intern->flags & SPL_DLLIST_IT_LIFO);

    if (element != NULL) {
      /* call dtor on the old element as in spl_ptr_llist_pop */
      if (intern->llist->dtor) {
        intern->llist->dtor(element);
      }

      /* the element is replaced, delref the old one as in
       * SplDoublyLinkedList::pop() */
      zval_ptr_dtor(&element->data);
      ZVAL_COPY_VALUE(&element->data, value);

      /* new element, call ctor as in spl_ptr_llist_push */
      if (intern->llist->ctor) {
        intern->llist->ctor(element);
      }
    } else {
      zval_ptr_dtor(value);
      zend_throw_exception(spl_ce_OutOfRangeException, "Offset invalid", 0);
      return;
    }
  }
} /* }}} */

/* {{{ proto void SplDoublyLinkedList::offsetUnset(mixed index)
 Unsets the value at the specified $index. */
SPL_METHOD(SplDoublyLinkedList, offsetUnset)
{
  zval                  *zindex;
  zend_long             index;
  spl_dllist_object     *intern;
  spl_ptr_llist_element *element;
  spl_ptr_llist         *llist;

  if (zend_parse_parameters(ZEND_NUM_ARGS(), "z", &zindex) == FAILURE) {
    return;
  }

  intern = Z_SPLDLLIST_P(getThis());
  index  = spl_offset_convert_to_long(zindex);
  llist  = intern->llist;

  if (index < 0 || index >= intern->llist->count) {
    zend_throw_exception(spl_ce_OutOfRangeException, "Offset out of range", 0);
    return;
  }

  element = spl_ptr_llist_offset(intern->llist, index, intern->flags & SPL_DLLIST_IT_LIFO);

  if (element != NULL) {
    /* connect the neightbors */
    if (element->prev) {
      element->prev->next = element->next;
    }

    if (element->next) {
      element->next->prev = element->prev;
    }

    /* take care of head/tail */
    if (element == llist->head) {
      llist->head = element->next;
    }

    if (element == llist->tail) {
      llist->tail = element->prev;
    }

    /* finally, delete the element */
    llist->count--;

    if(llist->dtor) {
      llist->dtor(element);
    }

    if (intern->traverse_pointer == element) {
      SPL_LLIST_DELREF(element);
      intern->traverse_pointer = NULL;
    }
    zval_ptr_dtor(&element->data);
    ZVAL_UNDEF(&element->data);

    SPL_LLIST_DELREF(element);
  } else {
    zend_throw_exception(spl_ce_OutOfRangeException, "Offset invalid", 0);
    return;
  }
} /* }}} */

static void spl_dllist_it_dtor(zend_object_iterator *iter) /* {{{ */
{
  spl_dllist_it *iterator = (spl_dllist_it *)iter;

  SPL_LLIST_CHECK_DELREF(iterator->traverse_pointer);

  zend_user_it_invalidate_current(iter);
  zval_ptr_dtor(&iterator->intern.it.data);
}
/* }}} */

static void spl_dllist_it_helper_rewind(spl_ptr_llist_element **traverse_pointer_ptr, int *traverse_position_ptr, spl_ptr_llist *llist, int flags) /* {{{ */
{
  SPL_LLIST_CHECK_DELREF(*traverse_pointer_ptr);

  if (flags & SPL_DLLIST_IT_LIFO) {
    *traverse_position_ptr = llist->count-1;
    *traverse_pointer_ptr  = llist->tail;
  } else {
    *traverse_position_ptr = 0;
    *traverse_pointer_ptr  = llist->head;
  }

  SPL_LLIST_CHECK_ADDREF(*traverse_pointer_ptr);
}
/* }}} */

static void spl_dllist_it_helper_move_forward(spl_ptr_llist_element **traverse_pointer_ptr, int *traverse_position_ptr, spl_ptr_llist *llist, int flags) /* {{{ */
{
  if (*traverse_pointer_ptr) {
    spl_ptr_llist_element *old = *traverse_pointer_ptr;

    if (flags & SPL_DLLIST_IT_LIFO) {
      *traverse_pointer_ptr = old->prev;
      (*traverse_position_ptr)--;

      if (flags & SPL_DLLIST_IT_DELETE) {
        zval prev;
        spl_ptr_llist_pop(llist, &prev);

        zval_ptr_dtor(&prev);
      }
    } else {
      *traverse_pointer_ptr = old->next;

      if (flags & SPL_DLLIST_IT_DELETE) {
        zval prev;
        spl_ptr_llist_shift(llist, &prev);

        zval_ptr_dtor(&prev);
      } else {
        (*traverse_position_ptr)++;
      }
    }

    SPL_LLIST_DELREF(old);
    SPL_LLIST_CHECK_ADDREF(*traverse_pointer_ptr);
  }
}
/* }}} */

static void spl_dllist_it_rewind(zend_object_iterator *iter) /* {{{ */
{
  spl_dllist_it *iterator = (spl_dllist_it *)iter;
  spl_dllist_object *object = Z_SPLDLLIST_P(&iter->data);
  spl_ptr_llist *llist = object->llist;

  spl_dllist_it_helper_rewind(&iterator->traverse_pointer, &iterator->traverse_position, llist, object->flags);
}
/* }}} */

static int spl_dllist_it_valid(zend_object_iterator *iter) /* {{{ */
{
  spl_dllist_it         *iterator = (spl_dllist_it *)iter;
  spl_ptr_llist_element *element  = iterator->traverse_pointer;

  return (element != NULL ? SUCCESS : FAILURE);
}
/* }}} */

static zval *spl_dllist_it_get_current_data(zend_object_iterator *iter) /* {{{ */
{
  spl_dllist_it         *iterator = (spl_dllist_it *)iter;
  spl_ptr_llist_element *element  = iterator->traverse_pointer;

  if (element == NULL || Z_ISUNDEF(element->data)) {
    return NULL;
  }

  return &element->data;
}
/* }}} */

static void spl_dllist_it_get_current_key(zend_object_iterator *iter, zval *key) /* {{{ */
{
  spl_dllist_it *iterator = (spl_dllist_it *)iter;

  ZVAL_LONG(key, iterator->traverse_position);
}
/* }}} */

static void spl_dllist_it_move_forward(zend_object_iterator *iter) /* {{{ */
{
  spl_dllist_it *iterator = (spl_dllist_it *)iter;
  spl_dllist_object *object = Z_SPLDLLIST_P(&iter->data);

  zend_user_it_invalidate_current(iter);

  spl_dllist_it_helper_move_forward(&iterator->traverse_pointer, &iterator->traverse_position, object->llist, object->flags);
}
/* }}} */

/* {{{  proto int SplDoublyLinkedList::key()
   Return current array key */
SPL_METHOD(SplDoublyLinkedList, key)
{
  spl_dllist_object *intern = Z_SPLDLLIST_P(getThis());

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  RETURN_LONG(intern->traverse_position);
}
/* }}} */

/* {{{ proto void SplDoublyLinkedList::prev()
   Move to next entry */
SPL_METHOD(SplDoublyLinkedList, prev)
{
  spl_dllist_object *intern = Z_SPLDLLIST_P(getThis());

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  spl_dllist_it_helper_move_forward(&intern->traverse_pointer, &intern->traverse_position, intern->llist, intern->flags ^ SPL_DLLIST_IT_LIFO);
}
/* }}} */

/* {{{ proto void SplDoublyLinkedList::next()
   Move to next entry */
SPL_METHOD(SplDoublyLinkedList, next)
{
  spl_dllist_object *intern = Z_SPLDLLIST_P(getThis());

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  spl_dllist_it_helper_move_forward(&intern->traverse_pointer, &intern->traverse_position, intern->llist, intern->flags);
}
/* }}} */

/* {{{ proto bool SplDoublyLinkedList::valid()
   Check whether the datastructure contains more entries */
SPL_METHOD(SplDoublyLinkedList, valid)
{
  spl_dllist_object *intern = Z_SPLDLLIST_P(getThis());

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  RETURN_BOOL(intern->traverse_pointer != NULL);
}
/* }}} */

/* {{{ proto void SplDoublyLinkedList::rewind()
   Rewind the datastructure back to the start */
SPL_METHOD(SplDoublyLinkedList, rewind)
{
  spl_dllist_object *intern = Z_SPLDLLIST_P(getThis());

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  spl_dllist_it_helper_rewind(&intern->traverse_pointer, &intern->traverse_position, intern->llist, intern->flags);
}
/* }}} */

/* {{{ proto mixed|NULL SplDoublyLinkedList::current()
   Return current datastructure entry */
SPL_METHOD(SplDoublyLinkedList, current)
{
  spl_dllist_object     *intern  = Z_SPLDLLIST_P(getThis());
  spl_ptr_llist_element *element = intern->traverse_pointer;

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  if (element == NULL || Z_ISUNDEF(element->data)) {
    RETURN_NULL();
  } else {
    zval *value = &element->data;

    ZVAL_DEREF(value);
    ZVAL_COPY(return_value, value);
  }
```

```c
}
/* }}} */

/* {{{ proto string SplDoublyLinkedList::serialize()
 Serializes storage */
SPL_METHOD(SplDoublyLinkedList, serialize)
{
  spl_dllist_object     *intern   = Z_SPLDLLIST_P(getThis());
  smart_str              buf      = {0};
  spl_ptr_llist_element *current  = intern->llist->head, *next;
  zval                   flags;
  php_serialize_data_t   var_hash;

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  PHP_VAR_SERIALIZE_INIT(var_hash);

  /* flags */
  ZVAL_LONG(&flags, intern->flags);
  php_var_serialize(&buf, &flags, &var_hash);
  zval_ptr_dtor(&flags);

  /* elements */
  while (current) {
    smart_str_appendc(&buf, ':');
    next = current->next;

    php_var_serialize(&buf, &current->data, &var_hash);

    current = next;
  }

  smart_str_0(&buf);

  /* done */
  PHP_VAR_SERIALIZE_DESTROY(var_hash);

  if (buf.s) {
    RETURN_NEW_STR(buf.s);
  } else {
    RETURN_NULL();
  }

} /* }}} */

/* {{{ proto void SplDoublyLinkedList::unserialize(string serialized)
 Unserializes storage */
SPL_METHOD(SplDoublyLinkedList, unserialize)
{
  spl_dllist_object *intern = Z_SPLDLLIST_P(getThis());
  zval *flags, *elem;
  char *buf;
  size_t buf_len;
  const unsigned char *p, *s;
  php_unserialize_data_t var_hash;

  if (zend_parse_parameters(ZEND_NUM_ARGS(), "s", &buf, &buf_len) == FAILURE) {
    return;
  }

  if (buf_len == 0) {
    return;
  }

  s = p = (const unsigned char*)buf;
  PHP_VAR_UNSERIALIZE_INIT(var_hash);

  /* flags */
  flags = var_tmp_var(&var_hash);
  if (!php_var_unserialize(flags, &p, s + buf_len, &var_hash) || Z_TYPE_P(flags) != IS_LONG) {
    goto error;
  }

  intern->flags = (int)Z_LVAL_P(flags);

  /* elements */
  while(*p == ':') {
    ++p;
    elem = var_tmp_var(&var_hash);
    if (!php_var_unserialize(elem, &p, s + buf_len, &var_hash)) {
      goto error;
    }
    var_push_dtor(&var_hash, elem);

    spl_ptr_llist_push(intern->llist, elem);
  }

  if (*p != '\0') {
    goto error;
  }

  PHP_VAR_UNSERIALIZE_DESTROY(var_hash);
  return;

error:
  PHP_VAR_UNSERIALIZE_DESTROY(var_hash);
  zend_throw_exception_ex(spl_ce_UnexpectedValueException, 0, "Error at offset %zd of %zd bytes", ((char*)p - buf), buf_len);
  return;

} /* }}} */

/* {{{ proto void SplDoublyLinkedList::add(mixed index, mixed newval)
 Inserts a new entry before the specified $index consisting of $newval. */
SPL_METHOD(SplDoublyLinkedList, add)
{
  zval                  *zindex, *value;
  spl_dllist_object     *intern;
  spl_ptr_llist_element *element;
  zend_long                  index;

  if (zend_parse_parameters(ZEND_NUM_ARGS(), "zz", &zindex, &value) == FAILURE) {
    return;
  }

  intern = Z_SPLDLLIST_P(getThis());
  index  = spl_offset_convert_to_long(zindex);

  if (index < 0 || index > intern->llist->count) {
    zend_throw_exception(spl_ce_OutOfRangeException, "Offset invalid or out of range", 0);
    return;
  }

  Z_TRY_ADDREF_P(value);
  if (index == intern->llist->count) {
    /* If index is the last entry+1 then we do a push because we're not inserting before any entry */
    spl_ptr_llist_push(intern->llist, value);
  } else {
    /* Create the new element we want to insert */
    spl_ptr_llist_element *elem = emalloc(sizeof(spl_ptr_llist_element));

    /* Get the element we want to insert before */
    element = spl_ptr_llist_offset(intern->llist, index, intern->flags & SPL_DLLIST_IT_LIFO);

    ZVAL_COPY_VALUE(&elem->data, value);
    elem->rc   = 1;
    /* connect to the neighbours */
    elem->next = element;
    elem->prev = element->prev;

    /* connect the neighbours to this new element */
    if (elem->prev == NULL) {
      intern->llist->head = elem;
    } else {
      element->prev->next = elem;
    }
    element->prev = elem;

    intern->llist->count++;

    if (intern->llist->ctor) {
      intern->llist->ctor(elem);
    }
  }
} /* }}} */

/* {{{ iterator handler table */
static const zend_object_iterator_funcs spl_dllist_it_funcs = {
  spl_dllist_it_dtor,
  spl_dllist_it_valid,
  spl_dllist_it_get_current_data,
  spl_dllist_it_get_current_key,
  spl_dllist_it_move_forward,
  spl_dllist_it_rewind,
  NULL
}; /* }}} */

zend_object_iterator *spl_dllist_get_iterator(zend_class_entry *ce, zval *object, int by_ref) /* {{{ */
{
  spl_dllist_it *iterator;
  spl_dllist_object *dllist_object = Z_SPLDLLIST_P(object);

  if (by_ref) {
    zend_throw_exception(spl_ce_RuntimeException, "An iterator cannot be used with foreach by reference", 0);
    return NULL;
  }

  iterator = emalloc(sizeof(spl_dllist_it));

  zend_iterator_init((zend_object_iterator*)iterator);

  ZVAL_COPY(&iterator->intern.it.data, object);
  iterator->intern.it.funcs    = &spl_dllist_it_funcs;
  iterator->intern.ce          = ce;
  iterator->traverse_position  = dllist_object->traverse_position;
  iterator->traverse_pointer   = dllist_object->traverse_pointer;
  iterator->flags              = dllist_object->flags & SPL_DLLIST_IT_MASK;
  ZVAL_UNDEF(&iterator->intern.value);

  SPL_LLIST_CHECK_ADDREF(iterator->traverse_pointer);

  return &iterator->intern.it;
}
/* }}} */

/*  Function/Class/Method definitions */
ZEND_BEGIN_ARG_INFO(arginfo_dllist_setiteratormode, 0)
  ZEND_ARG_INFO(0, flags)
ZEND_END_ARG_INFO()

ZEND_BEGIN_ARG_INFO(arginfo_dllist_push, 0)
  ZEND_ARG_INFO(0, value)
ZEND_END_ARG_INFO()

ZEND_BEGIN_ARG_INFO_EX(arginfo_dllist_offsetGet, 0, 0, 1)
  ZEND_ARG_INFO(0, index)
ZEND_END_ARG_INFO()

ZEND_BEGIN_ARG_INFO_EX(arginfo_dllist_offsetSet, 0, 0, 2)
  ZEND_ARG_INFO(0, index)
  ZEND_ARG_INFO(0, newval)
ZEND_END_ARG_INFO()

ZEND_BEGIN_ARG_INFO(arginfo_dllist_void, 0)
ZEND_END_ARG_INFO()

ZEND_BEGIN_ARG_INFO(arginfo_dllist_serialized, 0)
  ZEND_ARG_INFO(0, serialized)
ZEND_END_ARG_INFO();

static const zend_function_entry spl_funcs_SplQueue[] = {
  SPL_MA(SplQueue, enqueue, SplDoublyLinkedList, push,  arginfo_dllist_push, ZEND_ACC_PUBLIC)
  SPL_MA(SplQueue, dequeue, SplDoublyLinkedList, shift, arginfo_dllist_void, ZEND_ACC_PUBLIC)
  PHP_FE_END
};

static const zend_function_entry spl_funcs_SplDoublyLinkedList[] = {
  SPL_ME(SplDoublyLinkedList, pop,             arginfo_dllist_void,            ZEND_ACC_PUBLIC)
  SPL_ME(SplDoublyLinkedList, shift,           arginfo_dllist_void,            ZEND_ACC_PUBLIC)
  SPL_ME(SplDoublyLinkedList, push,            arginfo_dllist_push,            ZEND_ACC_PUBLIC)
  SPL_ME(SplDoublyLinkedList, unshift,         arginfo_dllist_push,            ZEND_ACC_PUBLIC)
  SPL_ME(SplDoublyLinkedList, top,             arginfo_dllist_void,            ZEND_ACC_PUBLIC)
  SPL_ME(SplDoublyLinkedList, bottom,          arginfo_dllist_void,            ZEND_ACC_PUBLIC)
  SPL_ME(SplDoublyLinkedList, isEmpty,         arginfo_dllist_void,            ZEND_ACC_PUBLIC)
  SPL_ME(SplDoublyLinkedList, setIteratorMode, arginfo_dllist_setiteratormode, ZEND_ACC_PUBLIC)
  SPL_ME(SplDoublyLinkedList, getIteratorMode, arginfo_dllist_void,            ZEND_ACC_PUBLIC)
  /* Countable */
  SPL_ME(SplDoublyLinkedList, count,           arginfo_dllist_void,            ZEND_ACC_PUBLIC)
  /* ArrayAccess */
  SPL_ME(SplDoublyLinkedList, offsetExists,    arginfo_dllist_offsetGet,       ZEND_ACC_PUBLIC)
  SPL_ME(SplDoublyLinkedList, offsetGet,       arginfo_dllist_offsetGet,       ZEND_ACC_PUBLIC)
  SPL_ME(SplDoublyLinkedList, offsetSet,       arginfo_dllist_offsetSet,       ZEND_ACC_PUBLIC)
  SPL_ME(SplDoublyLinkedList, offsetUnset,     arginfo_dllist_offsetGet,       ZEND_ACC_PUBLIC)

  SPL_ME(SplDoublyLinkedList, add,             arginfo_dllist_offsetSet,       ZEND_ACC_PUBLIC)

  /* Iterator */
  SPL_ME(SplDoublyLinkedList, rewind,          arginfo_dllist_void,            ZEND_ACC_PUBLIC)
  SPL_ME(SplDoublyLinkedList, current,         arginfo_dllist_void,            ZEND_ACC_PUBLIC)
  SPL_ME(SplDoublyLinkedList, key,             arginfo_dllist_void,            ZEND_ACC_PUBLIC)
  SPL_ME(SplDoublyLinkedList, next,            arginfo_dllist_void,            ZEND_ACC_PUBLIC)
  SPL_ME(SplDoublyLinkedList, prev,            arginfo_dllist_void,            ZEND_ACC_PUBLIC)
  SPL_ME(SplDoublyLinkedList, valid,           arginfo_dllist_void,            ZEND_ACC_PUBLIC)
  /* Serializable */
  SPL_ME(SplDoublyLinkedList,  unserialize,    arginfo_dllist_serialized,      ZEND_ACC_PUBLIC)
  SPL_ME(SplDoublyLinkedList,  serialize,      arginfo_dllist_void,            ZEND_ACC_PUBLIC)
  PHP_FE_END
};
/* }}} */

PHP_MINIT_FUNCTION(spl_dllist) /* {{{ */
{
  REGISTER_SPL_STD_CLASS_EX(SplDoublyLinkedList, spl_dllist_object_new, spl_funcs_SplDoublyLinkedList);
  memcpy(&spl_handler_SplDoublyLinkedList, zend_get_std_object_handlers(), sizeof(zend_object_handlers));

  spl_handler_SplDoublyLinkedList.offset = XtOffsetOf(spl_dllist_object, std);
  spl_handler_SplDoublyLinkedList.clone_obj = spl_dllist_object_clone;
  spl_handler_SplDoublyLinkedList.count_elements = spl_dllist_object_count_elements;
  spl_handler_SplDoublyLinkedList.get_debug_info = spl_dllist_object_get_debug_info;
  spl_handler_SplDoublyLinkedList.get_gc = spl_dllist_object_get_gc;
  spl_handler_SplDoublyLinkedList.dtor_obj = zend_objects_destroy_object;
  spl_handler_SplDoublyLinkedList.free_obj = spl_dllist_object_free_storage;

  REGISTER_SPL_CLASS_CONST_LONG(SplDoublyLinkedList, "IT_MODE_LIFO",  SPL_DLLIST_IT_LIFO);
  REGISTER_SPL_CLASS_CONST_LONG(SplDoublyLinkedList, "IT_MODE_FIFO",  0);
  REGISTER_SPL_CLASS_CONST_LONG(SplDoublyLinkedList, "IT_MODE_DELETE",SPL_DLLIST_IT_DELETE);
  REGISTER_SPL_CLASS_CONST_LONG(SplDoublyLinkedList, "IT_MODE_KEEP",  0);

  REGISTER_SPL_IMPLEMENTS(SplDoublyLinkedList, Iterator);
  REGISTER_SPL_IMPLEMENTS(SplDoublyLinkedList, Countable);
  REGISTER_SPL_IMPLEMENTS(SplDoublyLinkedList, ArrayAccess);
  REGISTER_SPL_IMPLEMENTS(SplDoublyLinkedList, Serializable);

  spl_ce_SplDoublyLinkedList->get_iterator = spl_dllist_get_iterator;

  REGISTER_SPL_SUB_CLASS_EX(SplQueue,           SplDoublyLinkedList,        spl_dllist_object_new, spl_funcs_SplQueue);
  REGISTER_SPL_SUB_CLASS_EX(SplStack,           SplDoublyLinkedList,        spl_dllist_object_new, NULL);

  spl_ce_SplQueue->get_iterator = spl_dllist_get_iterator;
  spl_ce_SplStack->get_iterator = spl_dllist_get_iterator;

  return SUCCESS;
}
/* }}} */

/*
 * Local variables:
 * tab-width: 4
 * c-basic-offset: 4
 * End:
 * vim600: fdm=marker
 * vim: noet sw=4 ts=4
 */
```

```c
/*
   +----------------------------------------------------------------------+
   | PHP Version 7                                                        |
   +----------------------------------------------------------------------+
   | Copyright (c) 1997-2018 The PHP Group                                |
   +----------------------------------------------------------------------+
   | This source file is subject to version 3.01 of the PHP license,      |
   | that is bundled with this package in the file LICENSE, and is        |
   | available through the world-wide-web at the following url:           |
   | http://www.php.net/license/3_01.txt                                  |
   | If you did not receive a copy of the PHP license and are unable to   |
   | obtain it through the world-wide-web, please send a note to          |
   | license@php.net so we can mail you a copy immediately.               |
   +----------------------------------------------------------------------+
   | Authors: Marcus Boerger <helly@php.net>                              |
   +----------------------------------------------------------------------+
 */

#ifdef HAVE_CONFIG_H
# include "config.h"
#endif

#include "php.h"
#include "php_ini.h"
#include "ext/standard/info.h"
#include "zend_interfaces.h"

#include "php_spl.h"
#include "spl_functions.h"
#include "spl_engine.h"

#include "spl_array.h"

/* {{{ spl_instantiate */
PHPAPI void spl_instantiate(zend_class_entry *pce, zval *object)
{
	object_init_ex(object, pce);
}
/* }}} */

PHPAPI zend_long spl_offset_convert_to_long(zval *offset) /* {{{ */
{
	zend_ulong idx;

try_again:
	switch (Z_TYPE_P(offset)) {
	case IS_STRING:
		if (ZEND_HANDLE_NUMERIC(Z_STR_P(offset), idx)) {
			return idx;
		}
		break;
	case IS_DOUBLE:
		return (zend_long)Z_DVAL_P(offset);
	case IS_LONG:
		return Z_LVAL_P(offset);
	case IS_FALSE:
		return 0;
	case IS_TRUE:
		return 1;
	case IS_REFERENCE:
		offset = Z_REFVAL_P(offset);
		goto try_again;
	case IS_RESOURCE:
		return Z_RES_HANDLE_P(offset);
	}
	return -1;
}
/* }}} */

/*
 * Local variables:
 * tab-width: 4
 * c-basic-offset: 4
 * End:
 * vim600: fdm=marker
 * vim: noet sw=4 ts=4
 */
```

```c
/*
   +----------------------------------------------------------------------+
   | PHP Version 7                                                        |
   +----------------------------------------------------------------------+
   | Copyright (c) 1997-2018 The PHP Group                                |
   +----------------------------------------------------------------------+
   | This source file is subject to version 3.01 of the PHP license,      |
   | that is bundled with this package in the file LICENSE, and is        |
   | available through the world-wide-web at the following url:           |
   | http://www.php.net/license/3_01.txt                                  |
   | If you did not receive a copy of the PHP license and are unable to   |
   | obtain it through the world-wide-web, please send a note to          |
   | license@php.net so we can mail you a copy immediately.               |
   +----------------------------------------------------------------------+
   | Author: Marcus Boerger <helly@php.net>                               |
   +----------------------------------------------------------------------+
 */

/* $Id$ */

#ifdef HAVE_CONFIG_H
# include "config.h"
#endif

#include "php.h"
#include "php_ini.h"
#include "ext/standard/info.h"
#include "ext/standard/file.h"
#include "ext/standard/php_string.h"
#include "zend_compile.h"
#include "zend_exceptions.h"
#include "zend_interfaces.h"

#include "php_spl.h"
#include "spl_functions.h"
#include "spl_engine.h"
#include "spl_iterators.h"
#include "spl_directory.h"
#include "spl_exceptions.h"

#include "php.h"
#include "fopen_wrappers.h"

#include "ext/standard/basic_functions.h"
#include "ext/standard/php_filestat.h"

#define SPL_HAS_FLAG(flags, test_flag) ((flags & test_flag) ? 1 : 0)

/* declare the class handlers */
static zend_object_handlers spl_filesystem_object_handlers;
/* includes handler to validate object state when retrieving methods */
static zend_object_handlers spl_filesystem_object_check_handlers;

/* decalre the class entry */
PHPAPI zend_class_entry *spl_ce_SplFileInfo;
PHPAPI zend_class_entry *spl_ce_DirectoryIterator;
PHPAPI zend_class_entry *spl_ce_FilesystemIterator;
PHPAPI zend_class_entry *spl_ce_RecursiveDirectoryIterator;
PHPAPI zend_class_entry *spl_ce_GlobIterator;
PHPAPI zend_class_entry *spl_ce_SplFileObject;
PHPAPI zend_class_entry *spl_ce_SplTempFileObject;

static void spl_filesystem_file_free_line(spl_filesystem_object *intern) /* {{{ */
{
  if (intern->u.file.current_line) {
    efree(intern->u.file.current_line);
    intern->u.file.current_line = NULL;
  }
  if (!Z_ISUNDEF(intern->u.file.current_zval)) {
    zval_ptr_dtor(&intern->u.file.current_zval);
    ZVAL_UNDEF(&intern->u.file.current_zval);
  }
} /* }}} */

static void spl_filesystem_object_destroy_object(zend_object *object) /* {{{ */
{
  spl_filesystem_object *intern = spl_filesystem_from_obj(object);

  zend_objects_destroy_object(object);

  switch(intern->type) {
  case SPL_FS_DIR:
    if (intern->u.dir.dirp) {
      php_stream_close(intern->u.dir.dirp);
      intern->u.dir.dirp = NULL;
    }
    break;
  case SPL_FS_FILE:
    if (intern->u.file.stream) {
      /*
      if (intern->u.file.zcontext) {
         zend_list_delref(Z_RESVAL_P(intern->zcontext));
      }
      */
      if (!intern->u.file.stream->is_persistent) {
        php_stream_close(intern->u.file.stream);
      } else {
        php_stream_pclose(intern->u.file.stream);
      }
    }
    break;
  default:
    break;
  }
} /* }}} */

static void spl_filesystem_object_free_storage(zend_object *object) /* {{{ */
{
  spl_filesystem_object *intern = spl_filesystem_from_obj(object);

  if (intern->oth_handler && intern->oth_handler->dtor) {
    intern->oth_handler->dtor(intern);
  }

  zend_object_std_dtor(&intern->std);

  if (intern->_path) {
    efree(intern->_path);
  }
  if (intern->file_name) {
    efree(intern->file_name);
  }
  switch(intern->type) {
  case SPL_FS_INFO:
    break;
  case SPL_FS_DIR:
    if (intern->u.dir.sub_path) {
      efree(intern->u.dir.sub_path);
    }
    break;
  case SPL_FS_FILE:
    if (intern->u.file.stream) {
      if (intern->u.file.open_mode) {
        efree(intern->u.file.open_mode);
      }
      if (intern->orig_path) {
        efree(intern->orig_path);
      }
    }
    spl_filesystem_file_free_line(intern);
    break;
  }
} /* }}} */

/* {{{ spl_ce_dir_object_new */
/* creates the object by
   - allocating memory
   - initializing the object members
   - storing the object
   - setting it’s handlers

   called from
   - clone
   - new
 */
static zend_object *spl_filesystem_object_new_ex(zend_class_entry *class_type)
{
  spl_filesystem_object *intern;

  intern = zend_object_alloc(sizeof(spl_filesystem_object), class_type);
  /* intern->type = SPL_FS_INFO; done by set 0 */
  intern->file_class = spl_ce_SplFileObject;
  intern->info_class = spl_ce_SplFileInfo;

  zend_object_std_init(&intern->std, class_type);
  object_properties_init(&intern->std, class_type);
  intern->std.handlers = &spl_filesystem_object_handlers;

  return &intern->std;
}
/* }}} */

/* {{{ spl_filesystem_object_new */
/* See spl_filesystem_object_new_ex */
static zend_object *spl_filesystem_object_new(zend_class_entry *class_type)
{
  return spl_filesystem_object_new_ex(class_type);
}
/* }}} */

/* {{{ spl_filesystem_object_new_check */
static zend_object *spl_filesystem_object_new_check(zend_class_entry *class_type)
{
  spl_filesystem_object *ret = spl_filesystem_from_obj(spl_filesystem_object_new_ex(class_type));
  ret->std.handlers = &spl_filesystem_object_check_handlers;
  return &ret->std;
}
/* }}} */

PHPAPI char* spl_filesystem_object_get_path(spl_filesystem_object *intern, size_t *len) /* {{{ */
{
#ifdef HAVE_GLOB
  if (intern->type == SPL_FS_DIR) {
    if (php_stream_is(intern->u.dir.dirp ,&php_glob_stream_ops)) {
      return php_glob_stream_get_path(intern->u.dir.dirp, 0, len);
    }
  }
#endif
  if (len) {
    *len = intern->_path_len;
  }
  return intern->_path;
} /* }}} */

static inline void spl_filesystem_object_get_file_name(spl_filesystem_object *intern) /* {{{ */
{
  char slash = SPL_HAS_FLAG(intern->flags, SPL_FILE_DIR_UNIXPATHS) ? '/' : DEFAULT_SLASH;

  switch (intern->type) {
    case SPL_FS_INFO:
    case SPL_FS_FILE:
      if (!intern->file_name) {
        php_error_docref(NULL, E_ERROR, "Object not initialized");
      }
      break;
    case SPL_FS_DIR:
      if (intern->file_name) {
        efree(intern->file_name);
      }
      intern->file_name_len = spprintf(&intern->file_name, 0, "%s%c%s",
                                       spl_filesystem_object_get_path(intern, NULL),
                                       slash, intern->u.dir.entry.d_name);
      break;
  }
} /* }}} */

static int spl_filesystem_dir_read(spl_filesystem_object *intern) /* {{{ */
{
  if (!intern->u.dir.dirp || !php_stream_readdir(intern->u.dir.dirp, &intern->u.dir.entry)) {
    intern->u.dir.entry.d_name[0] = '\0';
    return 0;
  } else {
    return 1;
  }
}
/* }}} */

#define IS_SLASH_AT(zs, pos) (IS_SLASH(zs[pos]))

static inline int spl_filesystem_is_dot(const char * d_name) /* {{{ */
{
  return !strcmp(d_name, ".") || !strcmp(d_name, "..");
}
/* }}} */

/* {{{ spl_filesystem_dir_open */
/* open a directory resource */
static void spl_filesystem_dir_open(spl_filesystem_object* intern, char *path)
{
  int skip_dots = SPL_HAS_FLAG(intern->flags, SPL_FILE_DIR_SKIPDOTS);

  intern->type = SPL_FS_DIR;
  intern->_path_len = strlen(path);
  intern->u.dir.dirp = php_stream_opendir(path, REPORT_ERRORS, FG(default_context));

  if (intern->_path_len > 1 && IS_SLASH_AT(path, intern->_path_len-1)) {
    intern->_path = estrndup(path, --intern->_path_len);
  } else {
    intern->_path = estrndup(path, intern->_path_len);
  }
  intern->u.dir.index = 0;

  if (EG(exception) || intern->u.dir.dirp == NULL) {
    intern->u.dir.entry.d_name[0] = '\0';
    if (!EG(exception)) {
      /* open failed w/out notice (turned to exception due to EH_THROW) */
      zend_throw_exception_ex(spl_ce_UnexpectedValueException, 0,
        "Failed to open directory \"%s\"", path);
    }
  } else {
    do {
      spl_filesystem_dir_read(intern);
    } while (skip_dots && spl_filesystem_is_dot(intern->u.dir.entry.d_name));
  }
}
/* }}} */

static int spl_filesystem_file_open(spl_filesystem_object *intern, int use_include_path, int silent) /* {{{ */
{
  zval tmp;

  intern->type = SPL_FS_FILE;

  php_stat(intern->file_name, intern->file_name_len, FS_IS_DIR, &tmp);
  if (Z_TYPE(tmp) == IS_TRUE) {
    intern->u.file.open_mode = NULL;
    intern->file_name = NULL;
    zend_throw_exception_ex(spl_ce_LogicException, 0, "Cannot use SplFileObject with directories");
    return FAILURE;
  }

  intern->u.file.context = php_stream_context_from_zval(intern->u.file.zcontext, 0);
  intern->u.file.stream = php_stream_open_wrapper_ex(intern->file_name, intern->u.file.open_mode, (use_include_path ? USE_PATH : 0) | REPORT_ERRORS, NULL, intern->u.file.context);

  if (!intern->file_name_len || !intern->u.file.stream) {
    if (!EG(exception)) {
      zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Cannot open file '%s'", intern->file_name_len ? intern->file_name : "");
    }
    intern->file_name = NULL; /* until here it is not a copy */
    intern->u.file.open_mode = NULL;
    return FAILURE;
  }

  /*
  if (intern->u.file.zcontext) {
    //zend_list_addref(Z_RES_VAL(intern->u.file.zcontext));
    Z_ADDREF_P(intern->u.file.zcontext);
  }
  */

  if (intern->file_name_len > 1 && IS_SLASH_AT(intern->file_name, intern->file_name_len-1)) {
    intern->file_name_len--;
  }

  intern->orig_path = estrndup(intern->u.file.stream->orig_path, strlen(intern->u.file.stream->orig_path));

  intern->file_name = estrndup(intern->file_name, intern->file_name_len);
  intern->u.file.open_mode = estrndup(intern->u.file.open_mode, intern->u.file.open_mode_len);

  /* avoid reference counting in debug mode, thus do it manually */
  ZVAL_RES(&intern->u.file.zresource, intern->u.file.stream->res);
  /*!!! TODO: maybe bug?
  Z_SET_REFCOUNT(intern->u.file.zresource, 1);
  */

  intern->u.file.delimiter = ',';
  intern->u.file.enclosure = '"';
  intern->u.file.escape = '\\';

  intern->u.file.func_getCurr = zend_hash_str_find_ptr(&intern->std.ce->function_table, "getcurrentline", sizeof("getcurrentline") - 1);

  return SUCCESS;
} /* }}} */

/* {{{ spl_filesystem_object_clone */
/* Local zend_object creation (on stack)
   Load the 'other' object
   Create a new empty object (See spl_filesystem_object_new_ex)
   Open the directory
   Clone other members (properties)
 */
static zend_object *spl_filesystem_object_clone(zval *zobject)
{
  zend_object *old_object;
  zend_object *new_object;
  spl_filesystem_object *intern;
  spl_filesystem_object *source;
  int index, skip_dots;

  old_object = Z_OBJ_P(zobject);
  source = spl_filesystem_from_obj(old_object);
  new_object = spl_filesystem_object_new_ex(old_object->ce);
  intern = spl_filesystem_from_obj(new_object);

  intern->flags = source->flags;

  switch (source->type) {
    case SPL_FS_INFO:
      intern->_path_len = source->_path_len;
      intern->_path = estrndup(source->_path, source->_path_len);
      intern->file_name_len = source->file_name_len;
      intern->file_name = estrndup(source->file_name, intern->file_name_len);
      break;
    case SPL_FS_DIR:
      spl_filesystem_dir_open(intern, source->_path);
      /* read until we hit the position in which we were before */
      skip_dots = SPL_HAS_FLAG(source->flags, SPL_FILE_DIR_SKIPDOTS);
      for(index = 0; index < source->u.dir.index; ++index) {
        do {
          spl_filesystem_dir_read(intern);
        } while (skip_dots && spl_filesystem_is_dot(intern->u.dir.entry.d_name));
      }
      intern->u.dir.index = index;
      break;
    case SPL_FS_FILE:
      zend_throw_error(NULL, "An object of class %s cannot be cloned", ZSTR_VAL(old_object->ce->name));
      return new_object;
  }

  intern->file_class = source->file_class;
  intern->info_class = source->info_class;
  intern->oth = source->oth;
  intern->oth_handler = source->oth_handler;

  zend_objects_clone_members(new_object, old_object);

  if (intern->oth_handler && intern->oth_handler->clone) {
    intern->oth_handler->clone(source, intern);
  }

  return new_object;
}
/* }}} */

void spl_filesystem_info_set_filename(spl_filesystem_object *intern, char *path, size_t len, size_t use_copy) /* {{{ */
{
  char *p1, *p2;

  if (intern->file_name) {
    efree(intern->file_name);
  }

  intern->file_name = use_copy ? estrndup(path, len) : path;
  intern->file_name_len = len;

  while (intern->file_name_len > 1 && IS_SLASH_AT(intern->file_name, intern->file_name_len-1)) {
    intern->file_name[intern->file_name_len-1] = 0;
    intern->file_name_len--;
  }

  p1 = strrchr(intern->file_name, '/');
#if defined(PHP_WIN32)
  p2 = strrchr(intern->file_name, '\\');
#else
  p2 = 0;
#endif
  if (p1 || p2) {
    intern->_path_len = ((p1 > p2 ? p1 : p2) - intern->file_name);
  } else {
    intern->_path_len = 0;
  }

  if (intern->_path) {
    efree(intern->_path);
  }
  intern->_path = estrndup(path, intern->_path_len);
} /* }}} */

static spl_filesystem_object *spl_filesystem_object_create_info(spl_filesystem_object *source, char *file_path, size_t file_path_len, int use_copy, zend_class_entry *ce, zval *return_value) /* {{{ */
{
  spl_filesystem_object *intern;
  zval arg1;
  zend_error_handling error_handling;

  if (!file_path || !file_path_len) {
#if defined(PHP_WIN32)
    zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Cannot create SplFileInfo for empty path");
    if (file_path && !use_copy) {
      efree(file_path);
    }
#else
    if (file_path && !use_copy) {
      efree(file_path);
    }
    file_path_len = 1;
    file_path = "/";
#endif
    return NULL;
  }

  zend_replace_error_handling(EH_THROW, spl_ce_RuntimeException, &error_handling);

  ce = ce ? ce : source->info_class;

  zend_update_class_constants(ce);

  intern = spl_filesystem_from_obj(spl_filesystem_object_new_ex(ce));
  ZVAL_OBJ(return_value, &intern->std);

  if (ce->constructor->common.scope != spl_ce_SplFileInfo) {
    ZVAL_STRINGL(&arg1, file_path, file_path_len);
    zend_call_method_with_1_params(return_value, ce, &ce->constructor, "__construct", NULL, &arg1);
    zval_ptr_dtor(&arg1);
  } else {
    spl_filesystem_info_set_filename(intern, file_path, file_path_len, use_copy);
  }

  zend_restore_error_handling(&error_handling);
  return intern;
} /* }}} */

static spl_filesystem_object *spl_filesystem_object_create_type(int ht, spl_filesystem_object *source, int type, zend_class_entry *ce, zval *return_value) /* {{{ */
{
  spl_filesystem_object *intern;
  zend_bool use_include_path = 0;
  zval arg1, arg2;
  zend_error_handling error_handling;

  zend_replace_error_handling(EH_THROW, spl_ce_RuntimeException, &error_handling);

  switch (source->type) {
    case SPL_FS_INFO:
    case SPL_FS_FILE:
      break;
    case SPL_FS_DIR:
      if (!source->u.dir.entry.d_name[0]) {
        zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Could not open file");
        zend_restore_error_handling(&error_handling);
        return NULL;
      }
  }

  switch (type) {
    case SPL_FS_INFO:
      ce = ce ? ce : source->info_class;

      if (UNEXPECTED(zend_update_class_constants(ce) != SUCCESS)) {
        break;
      }

      intern = spl_filesystem_from_obj(spl_filesystem_object_new_ex(ce));
      ZVAL_OBJ(return_value, &intern->std);

      spl_filesystem_object_get_file_name(source);
      if (ce->constructor->common.scope != spl_ce_SplFileInfo) {
        ZVAL_STRINGL(&arg1, source->file_name, source->file_name_len);
        zend_call_method_with_1_params(return_value, ce, &ce->constructor, "__construct", NULL, &arg1);
        zval_ptr_dtor(&arg1);
      } else {
        intern->file_name = estrndup(source->file_name, source->file_name_len);
        intern->file_name_len = source->file_name_len;
        intern->_path = spl_filesystem_object_get_path(source, &intern->_path_len);
        intern->_path = estrndup(intern->_path, intern->_path_len);
      }
      break;
    case SPL_FS_FILE:
      ce = ce ? ce : source->file_class;

      if (UNEXPECTED(zend_update_class_constants(ce) != SUCCESS)) {
        break;
      }

      intern = spl_filesystem_from_obj(spl_filesystem_object_new_ex(ce));

      ZVAL_OBJ(return_value, &intern->std);

      spl_filesystem_object_get_file_name(source);

      if (ce->constructor->common.scope != spl_ce_SplFileObject) {
        ZVAL_STRINGL(&arg1, source->file_name, source->file_name_len);
        ZVAL_STRINGL(&arg2, "r", 1);
        zend_call_method_with_2_params(return_value, ce, &ce->constructor, "__construct", NULL, &arg1, &arg2);
        zval_ptr_dtor(&arg1);
        zval_ptr_dtor(&arg2);
      } else {
        intern->file_name = source->file_name;
        intern->file_name_len = source->file_name_len;
        intern->_path = spl_filesystem_object_get_path(source, &intern->_path_len);
        intern->_path = estrndup(intern->_path, intern->_path_len);

        intern->u.file.open_mode = "r";
        intern->u.file.open_mode_len = 1;

        if (ht && zend_parse_parameters(ht, "|sbr",
              &intern->u.file.open_mode, &intern->u.file.open_mode_len,
              &use_include_path, &intern->u.file.zcontext) == FAILURE) {
          zend_restore_error_handling(&error_handling);
          intern->u.file.open_mode = NULL;
          intern->file_name = NULL;
          zval_ptr_dtor(return_value);
          ZVAL_NULL(return_value);
          return NULL;
        }

        if (spl_filesystem_file_open(intern, use_include_path, 0) == FAILURE) {
          zend_restore_error_handling(&error_handling);
          zval_ptr_dtor(return_value);
          ZVAL_NULL(return_value);
          return NULL;
        }
      }
      break;
    case SPL_FS_DIR:
      zend_restore_error_handling(&error_handling);
      zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Operation not supported");
      return NULL;
  }
  zend_restore_error_handling(&error_handling);
  return NULL;
} /* }}} */

static int spl_filesystem_is_invalid_or_dot(const char * d_name) /* {{{ */
{
  return d_name[0] == '\0' || spl_filesystem_is_dot(d_name);
}
/* }}} */

static char *spl_filesystem_object_get_pathname(spl_filesystem_object *intern, size_t *len) { /* {{{ */
  switch (intern->type) {
    case SPL_FS_INFO:
    case SPL_FS_FILE:
      *len = intern->file_name_len;
      return intern->file_name;
    case SPL_FS_DIR:
      if (intern->u.dir.entry.d_name[0]) {
        spl_filesystem_object_get_file_name(intern);
        *len = intern->file_name_len;
        return intern->file_name;
      }
  }
  *len = 0;
  return NULL;
}
/* }}} */

static HashTable *spl_filesystem_object_get_debug_info(zval *object, int *is_temp) /* {{{ */
{
  spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(object);
  zval tmp;
  HashTable *rv;
  zend_string *pnstr;
  char *path;
  size_t  path_len;
  char stmp[2];

  *is_temp = 1;

  if (!intern->std.properties) {
    rebuild_object_properties(&intern->std);
  }

  rv = zend_array_dup(intern->std.properties);

  pnstr = spl_gen_private_prop_name(spl_ce_SplFileInfo, "pathName", sizeof("pathName")-1);
  path = spl_filesystem_object_get_pathname(intern, &path_len);
  ZVAL_STRINGL(&tmp, path, path_len);
  zend_symtable_update(rv, pnstr, &tmp);
  zend_string_release(pnstr);

  if (intern->file_name) {
    pnstr = spl_gen_private_prop_name(spl_ce_SplFileInfo, "fileName", sizeof("fileName")-1);
    spl_filesystem_object_get_path(intern, &path_len);

    if (path_len && path_len < intern->file_name_len) {
      ZVAL_STRINGL(&tmp, intern->file_name + path_len + 1, intern->file_name_len - (path_len + 1));
    } else {
      ZVAL_STRINGL(&tmp, intern->file_name, intern->file_name_len);
    }
    zend_symtable_update(rv, pnstr, &tmp);
    zend_string_release(pnstr);
  }
  if (intern->type == SPL_FS_DIR) {
#ifdef HAVE_GLOB
    pnstr = spl_gen_private_prop_name(spl_ce_DirectoryIterator, "glob", sizeof("glob")-1);
    if (php_stream_is(intern->u.dir.dirp ,&php_glob_stream_ops)) {
      ZVAL_STRINGL(&tmp, intern->_path, intern->_path_len);
    } else {
      ZVAL_FALSE(&tmp);
    }
    zend_symtable_update(rv, pnstr, &tmp);
    zend_string_release(pnstr);
#endif
    pnstr = spl_gen_private_prop_name(spl_ce_RecursiveDirectoryIterator, "subPathName", sizeof("subPathName")-1);
    if (intern->u.dir.sub_path) {
      ZVAL_STRINGL(&tmp, intern->u.dir.sub_path, intern->u.dir.sub_path_len);
    } else {
      ZVAL_EMPTY_STRING(&tmp);
    }
    zend_symtable_update(rv, pnstr, &tmp);
    zend_string_release(pnstr);
  }
  if (intern->type == SPL_FS_FILE) {
    pnstr = spl_gen_private_prop_name(spl_ce_SplFileObject, "openMode", sizeof("openMode")-1);
    ZVAL_STRINGL(&tmp, intern->u.file.open_mode, intern->u.file.open_mode_len);
    zend_symtable_update(rv, pnstr, &tmp);
    zend_string_release(pnstr);
    stmp[1] = '\0';
    stmp[0] = intern->u.file.delimiter;
    pnstr = spl_gen_private_prop_name(spl_ce_SplFileObject, "delimiter", sizeof("delimiter")-1);
    ZVAL_STRINGL(&tmp, stmp, 1);
    zend_symtable_update(rv, pnstr, &tmp);
    zend_string_release(pnstr);
    stmp[0] = intern->u.file.enclosure;
    pnstr = spl_gen_private_prop_name(spl_ce_SplFileObject, "enclosure", sizeof("enclosure")-1);
    ZVAL_STRINGL(&tmp, stmp, 1);
    zend_symtable_update(rv, pnstr, &tmp);
    zend_string_release(pnstr);
  }

  return rv;
}
/* }}} */

zend_function *spl_filesystem_object_get_method_check(zend_object **object, zend_string *method, const zval *key) /* {{{ */
{
  spl_filesystem_object *fsobj = spl_filesystem_from_obj(*object);

  if (fsobj->u.dir.dirp == NULL && fsobj->orig_path == NULL) {
    zend_function *func;
    zend_string *tmp = zend_string_init("_bad_state_ex", sizeof("_bad_state_ex") - 1, 0);
    func = zend_get_std_object_handlers()->get_method(object, tmp, NULL);
    zend_string_release(tmp);
    return func;
  }

  return zend_get_std_object_handlers()->get_method(object, method, key);
}
/* }}} */

#define DIT_CTOR_FLAGS  0x00000001
#define DIT_CTOR_GLOB   0x00000002

void spl_filesystem_object_construct(INTERNAL_FUNCTION_PARAMETERS, zend_long ctor_flags) /* {{{ */
{
  spl_filesystem_object *intern;
  char *path;
  int parsed;
  size_t len;
  zend_long flags;
  zend_error_handling error_handling;

  zend_replace_error_handling(EH_THROW, spl_ce_UnexpectedValueException, &error_handling);

  if (SPL_HAS_FLAG(ctor_flags, DIT_CTOR_FLAGS)) {
    flags = SPL_FILE_DIR_KEY_AS_PATHNAME|SPL_FILE_DIR_CURRENT_AS_FILEINFO;
    parsed = zend_parse_parameters(ZEND_NUM_ARGS(), "s|l", &path, &len, &flags);
  } else {
    flags = SPL_FILE_DIR_KEY_AS_PATHNAME|SPL_FILE_DIR_CURRENT_AS_SELF;
    parsed = zend_parse_parameters(ZEND_NUM_ARGS(), "s", &path, &len);
  }
  if (SPL_HAS_FLAG(ctor_flags, SPL_FILE_DIR_SKIPDOTS)) {
    flags |= SPL_FILE_DIR_SKIPDOTS;
  }
  if (SPL_HAS_FLAG(ctor_flags, SPL_FILE_DIR_UNIXPATHS)) {
    flags |= SPL_FILE_DIR_UNIXPATHS;
  }
  if (parsed == FAILURE) {
    zend_restore_error_handling(&error_handling);
    return;
  }
  if (!len) {
    zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Directory name must not be empty.");
    zend_restore_error_handling(&error_handling);
    return;
  }

  intern = Z_SPLFILESYSTEM_P(getThis());
  if (intern->_path) {
    /* object is alreay initialized */
    zend_restore_error_handling(&error_handling);
    php_error_docref(NULL, E_WARNING, "Directory object is already initialized");
    return;
  }
  intern->flags = flags;
#ifdef HAVE_GLOB
  if (SPL_HAS_FLAG(ctor_flags, DIT_CTOR_GLOB) && strstr(path, "glob://") != path) {
    spprintf(&path, 0, "glob://%s", path);
    spl_filesystem_dir_open(intern, path);
    efree(path);
  } else
#endif
  {
    spl_filesystem_dir_open(intern, path);

  }

  intern->u.dir.is_recursive = instanceof_function(intern->std.ce, spl_ce_RecursiveDirectoryIterator) ? 1 : 0;

  zend_restore_error_handling(&error_handling);
```

```c
}
/* }}} */

/* {{{ proto void DirectoryIterator::__construct(string path)
 Cronstructs a new dir iterator from a path. */
SPL_METHOD(DirectoryIterator, __construct)
{
  spl_filesystem_object_construct(INTERNAL_FUNCTION_PARAM_PASSTHRU, 0);
}
/* }}} */

/* {{{ proto void DirectoryIterator::rewind()
   Rewind dir back to the start */
SPL_METHOD(DirectoryIterator, rewind)
{
  spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  intern->u.dir.index = 0;
  if (intern->u.dir.dirp) {
    php_stream_rewinddir(intern->u.dir.dirp);
  }
  spl_filesystem_dir_read(intern);
}
/* }}} */

/* {{{ proto string DirectoryIterator::key()
   Return current dir entry */
SPL_METHOD(DirectoryIterator, key)
{
  spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  if (intern->u.dir.dirp) {
    RETURN_LONG(intern->u.dir.index);
  } else {
    RETURN_FALSE;
  }
}
/* }}} */

/* {{{ proto DirectoryIterator DirectoryIterator::current()
   Return this (needed for Iterator interface) */
SPL_METHOD(DirectoryIterator, current)
{
  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }
  ZVAL_OBJ(return_value, Z_OBJ_P(getThis()));
  Z_ADDREF_P(return_value);
}
/* }}} */

/* {{{ proto void DirectoryIterator::next()
   Move to next entry */
SPL_METHOD(DirectoryIterator, next)
{
  spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());
  int skip_dots = SPL_HAS_FLAG(intern->flags, SPL_FILE_DIR_SKIPDOTS);

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  intern->u.dir.index++;
  do {
    spl_filesystem_dir_read(intern);
  } while (skip_dots && spl_filesystem_is_dot(intern->u.dir.entry.d_name));
  if (intern->file_name) {
    efree(intern->file_name);
    intern->file_name = NULL;
  }
}
/* }}} */

/* {{{ proto void DirectoryIterator::seek(int position)
   Seek to the given position */
SPL_METHOD(DirectoryIterator, seek)
{
  spl_filesystem_object *intern    = Z_SPLFILESYSTEM_P(getThis());
  zval retval;
  zend_long pos;

  if (zend_parse_parameters(ZEND_NUM_ARGS(), "l", &pos) == FAILURE) {
    return;
  }

  if (intern->u.dir.index > pos) {
    /* we first rewind */
    zend_call_method_with_0_params(&EX(This), Z_OBJCE(EX(This)), &intern->u.dir.func_rewind, "rewind", NULL);
  }

  while (intern->u.dir.index < pos) {
    int valid = 0;
    zend_call_method_with_0_params(&EX(This), Z_OBJCE(EX(This)), &intern->u.dir.func_valid, "valid", &retval);
    if (!Z_ISUNDEF(retval)) {
      valid = zend_is_true(&retval);
      zval_ptr_dtor(&retval);
    }
    if (!valid) {
      zend_throw_exception_ex(spl_ce_OutOfBoundsException, 0, "Seek position " ZEND_LONG_FMT " is out of range", pos);
      return;
    }
    zend_call_method_with_0_params(&EX(This), Z_OBJCE(EX(This)), &intern->u.dir.func_next, "next", NULL);
  }
} /* }}} */

/* {{{ proto string DirectoryIterator::valid()
   Check whether dir contains more entries */
SPL_METHOD(DirectoryIterator, valid)
{
  spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  RETURN_BOOL(intern->u.dir.entry.d_name[0] != '\0');
}
/* }}} */

/* {{{ proto string SplFileInfo::getPath()
   Return the path */
SPL_METHOD(SplFileInfo, getPath)
{
  spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());
  char *path;
  size_t path_len;

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

    path = spl_filesystem_object_get_path(intern, &path_len);
  RETURN_STRINGL(path, path_len);
}
/* }}} */

/* {{{ proto string SplFileInfo::getFilename()
   Return filename only */
SPL_METHOD(SplFileInfo, getFilename)
{
  spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());
  size_t path_len;

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  spl_filesystem_object_get_path(intern, &path_len);

  if (path_len && path_len < intern->file_name_len) {
    RETURN_STRINGL(intern->file_name + path_len + 1, intern->file_name_len - (path_len + 1));
  } else {
    RETURN_STRINGL(intern->file_name, intern->file_name_len);
  }
}
/* }}} */

/* {{{ proto string DirectoryIterator::getFilename()
   Return filename of current dir entry */
SPL_METHOD(DirectoryIterator, getFilename)
{
  spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  RETURN_STRING(intern->u.dir.entry.d_name);
}
/* }}} */

/* {{{ proto string SplFileInfo::getExtension()
   Returns file extension component of path */
SPL_METHOD(SplFileInfo, getExtension)
{
  spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());
  char *fname = NULL;
  const char *p;
  size_t flen;
  size_t path_len;
  size_t idx;
  zend_string *ret;

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  spl_filesystem_object_get_path(intern, &path_len);

  if (path_len && path_len < intern->file_name_len) {
    fname = intern->file_name + path_len + 1;
    flen = intern->file_name_len - (path_len + 1);
  } else {
    fname = intern->file_name;
    flen = intern->file_name_len;
  }

  ret = php_basename(fname, flen, NULL, 0);

  p = zend_memrchr(ZSTR_VAL(ret), '.', ZSTR_LEN(ret));
  if (p) {
    idx = p - ZSTR_VAL(ret);
    RETVAL_STRINGL(ZSTR_VAL(ret) + idx + 1, ZSTR_LEN(ret) - idx - 1);
    zend_string_release(ret);
    return;
  } else {
    zend_string_release(ret);
    RETURN_EMPTY_STRING();
  }
}
/* }}}*/

/* {{{ proto string DirectoryIterator::getExtension()
   Returns the file extension component of path */
SPL_METHOD(DirectoryIterator, getExtension)
{
  spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());
  const char *p;
  size_t idx;
  zend_string *fname;

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  fname = php_basename(intern->u.dir.entry.d_name, strlen(intern->u.dir.entry.d_name), NULL, 0);

  p = zend_memrchr(ZSTR_VAL(fname), '.', ZSTR_LEN(fname));
  if (p) {
    idx = p - ZSTR_VAL(fname);
    RETVAL_STRINGL(ZSTR_VAL(fname) + idx + 1, ZSTR_LEN(fname) - idx - 1);
    zend_string_release(fname);
  } else {
    zend_string_release(fname);
    RETURN_EMPTY_STRING();
  }
}
/* }}} */

/* {{{ proto string SplFileInfo::getBasename([string $suffix])
   Returns filename component of path */
SPL_METHOD(SplFileInfo, getBasename)
{
  spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());
  char *fname, *suffix = 0;
  size_t flen;
  size_t slen = 0, path_len;

  if (zend_parse_parameters(ZEND_NUM_ARGS(), "|s", &suffix, &slen) == FAILURE) {
    return;
  }

  spl_filesystem_object_get_path(intern, &path_len);

  if (path_len && path_len < intern->file_name_len) {
    fname = intern->file_name + path_len + 1;
    flen = intern->file_name_len - (path_len + 1);
  } else {
    fname = intern->file_name;
    flen = intern->file_name_len;
  }

  RETURN_STR(php_basename(fname, flen, suffix, slen));
}
/* }}}*/

/* {{{ proto string DirectoryIterator::getBasename([string $suffix])
   Returns filename component of current dir entry */
SPL_METHOD(DirectoryIterator, getBasename)
{
  spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());
  char *suffix = 0;
  size_t slen = 0;
  zend_string *fname;

  if (zend_parse_parameters(ZEND_NUM_ARGS(), "|s", &suffix, &slen) == FAILURE) {
    return;
  }

  fname = php_basename(intern->u.dir.entry.d_name, strlen(intern->u.dir.entry.d_name), suffix, slen);

  RETVAL_STR(fname);
}
/* }}} */

/* {{{ proto string SplFileInfo::getPathname()
   Return path and filename */
SPL_METHOD(SplFileInfo, getPathname)
{
  spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());
  char *path;
  size_t path_len;

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }
  path = spl_filesystem_object_get_pathname(intern, &path_len);
  if (path != NULL) {
    RETURN_STRINGL(path, path_len);
  } else {
    RETURN_FALSE;
  }
}
/* }}} */

/* {{{ proto string FilesystemIterator::key()
   Return getPathname() or getFilename() depending on flags */
SPL_METHOD(FilesystemIterator, key)
{
  spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  if (SPL_FILE_DIR_KEY(intern, SPL_FILE_DIR_KEY_AS_FILENAME)) {
    RETURN_STRING(intern->u.dir.entry.d_name);
  } else {
    spl_filesystem_object_get_file_name(intern);
    RETURN_STRINGL(intern->file_name, intern->file_name_len);
  }
}
/* }}} */

/* {{{ proto string FilesystemIterator::current()
   Return getFilename(), getFileInfo() or $this depending on flags */
SPL_METHOD(FilesystemIterator, current)
{
  spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  if (SPL_FILE_DIR_CURRENT(intern, SPL_FILE_DIR_CURRENT_AS_PATHNAME)) {
    spl_filesystem_object_get_file_name(intern);
    RETURN_STRINGL(intern->file_name, intern->file_name_len);
  } else if (SPL_FILE_DIR_CURRENT(intern, SPL_FILE_DIR_CURRENT_AS_FILEINFO)) {
    spl_filesystem_object_get_file_name(intern);
    spl_filesystem_object_create_type(0, intern, SPL_FS_INFO, NULL, return_value);
  } else {
    ZVAL_OBJ(return_value, Z_OBJ_P(getThis()));
    Z_ADDREF_P(return_value);
    /*RETURN_STRING(intern->u.dir.entry.d_name, 1);*/
  }
}
/* }}} */

/* {{{ proto bool DirectoryIterator::isDot()
   Returns true if current entry is '.' or  '..' */
SPL_METHOD(DirectoryIterator, isDot)
{
  spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  RETURN_BOOL(spl_filesystem_is_dot(intern->u.dir.entry.d_name));
}
/* }}} */

/* {{{ proto void SplFileInfo::__construct(string file_name)
 Cronstructs a new SplFileInfo from a path. */
/* When the constructor gets called the object is already created
   by the engine, so we must only call 'additional' initializations.
 */
SPL_METHOD(SplFileInfo, __construct)
```

```c
{
  spl_filesystem_object *intern;
  char *path;
  size_t len;

  if (zend_parse_parameters_throw(ZEND_NUM_ARGS(), "s", &path, &len) == FAILURE) {
    return;
  }

  intern = Z_SPLFILESYSTEM_P(getThis());

  spl_filesystem_info_set_filename(intern, path, len, 1);

  /* intern->type = SPL_FS_INFO; already set */
}
/* }}} */

/* {{{ FileInfoFunction */
#define FileInfoFunction(func_name, func_num) \
SPL_METHOD(SplFileInfo, func_name) \
{ \
  spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis()); \
  zend_error_handling error_handling; \
  if (zend_parse_parameters_none() == FAILURE) { \
    return; \
  } \
 \
  zend_replace_error_handling(EH_THROW, spl_ce_RuntimeException, &error_handling);\
  spl_filesystem_object_get_file_name(intern); \
  php_stat(intern->file_name, intern->file_name_len, func_num, return_value); \
  zend_restore_error_handling(&error_handling); \
}
/* }}} */

/* {{{ proto int SplFileInfo::getPerms()
   Get file permissions */
FileInfoFunction(getPerms, FS_PERMS)
/* }}} */

/* {{{ proto int SplFileInfo::getInode()
   Get file inode */
FileInfoFunction(getInode, FS_INODE)
/* }}} */

/* {{{ proto int SplFileInfo::getSize()
   Get file size */
FileInfoFunction(getSize, FS_SIZE)
/* }}} */

/* {{{ proto int SplFileInfo::getOwner()
   Get file owner */
FileInfoFunction(getOwner, FS_OWNER)
/* }}} */

/* {{{ proto int SplFileInfo::getGroup()
   Get file group */
FileInfoFunction(getGroup, FS_GROUP)
/* }}} */

/* {{{ proto int SplFileInfo::getATime()
   Get last access time of file */
FileInfoFunction(getATime, FS_ATIME)
/* }}} */

/* {{{ proto int SplFileInfo::getMTime()
   Get last modification time of file */
FileInfoFunction(getMTime, FS_MTIME)
/* }}} */

/* {{{ proto int SplFileInfo::getCTime()
   Get inode modification time of file */
FileInfoFunction(getCTime, FS_CTIME)
/* }}} */

/* {{{ proto string SplFileInfo::getType()
   Get file type */
FileInfoFunction(getType, FS_TYPE)
/* }}} */

/* {{{ proto bool SplFileInfo::isWritable()
   Returns true if file can be written */
FileInfoFunction(isWritable, FS_IS_W)
/* }}} */

/* {{{ proto bool SplFileInfo::isReadable()
   Returns true if file can be read */
FileInfoFunction(isReadable, FS_IS_R)
/* }}} */

/* {{{ proto bool SplFileInfo::isExecutable()
   Returns true if file is executable */
FileInfoFunction(isExecutable, FS_IS_X)
/* }}} */

/* {{{ proto bool SplFileInfo::isFile()
   Returns true if file is a regular file */
FileInfoFunction(isFile, FS_IS_FILE)
/* }}} */

/* {{{ proto bool SplFileInfo::isDir()
   Returns true if file is directory */
FileInfoFunction(isDir, FS_IS_DIR)
/* }}} */

/* {{{ proto bool SplFileInfo::isLink()
   Returns true if file is symbolic link */
FileInfoFunction(isLink, FS_IS_LINK)
/* }}} */

/* {{{ proto string SplFileInfo::getLinkTarget()
   Return the target of a symbolic link */
SPL_METHOD(SplFileInfo, getLinkTarget)
{
  spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());
  ssize_t ret;
  char buff[MAXPATHLEN];
  zend_error_handling error_handling;

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  zend_replace_error_handling(EH_THROW, spl_ce_RuntimeException, &error_handling);

#if defined(PHP_WIN32) || HAVE_SYMLINK
  if (intern->file_name == NULL) {
    php_error_docref(NULL, E_WARNING, "Empty filename");
    RETURN_FALSE;
  } else if (!IS_ABSOLUTE_PATH(intern->file_name, intern->file_name_len)) {
    char expanded_path[MAXPATHLEN];
    if (!expand_filepath_with_mode(intern->file_name, expanded_path, NULL, 0, CWD_EXPAND )) {
      php_error_docref(NULL, E_WARNING, "No such file or directory");
      RETURN_FALSE;
    }
    ret = php_sys_readlink(expanded_path, buff, MAXPATHLEN - 1);
  } else {
    ret = php_sys_readlink(intern->file_name, buff,  MAXPATHLEN-1);
  }
#else
  ret = -1; /* always fail if not implemented */
#endif

  if (ret == -1) {
    zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Unable to read link %s, error: %s", intern->file_name, strerror(errno));
    RETVAL_FALSE;
  } else {
    /* Append NULL to the end of the string */
    buff[ret] = '\0';

    RETVAL_STRINGL(buff, ret);
  }

  zend_restore_error_handling(&error_handling);
}
/* }}} */

#if HAVE_REALPATH || defined(ZTS)
/* {{{ proto string SplFileInfo::getRealPath()
   Return the resolved path */
SPL_METHOD(SplFileInfo, getRealPath)
{
  spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());
  char buff[MAXPATHLEN];
  char *filename;
  zend_error_handling error_handling;

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  zend_replace_error_handling(EH_THROW, spl_ce_RuntimeException, &error_handling);

  if (intern->type == SPL_FS_DIR && !intern->file_name && intern->u.dir.entry.d_name[0]) {
    spl_filesystem_object_get_file_name(intern);
  }

  if (intern->orig_path) {
    filename = intern->orig_path;
  } else {
    filename = intern->file_name;
  }


  if (filename && VCWD_REALPATH(filename, buff)) {
#ifdef ZTS
    if (VCWD_ACCESS(buff, F_OK)) {
      RETVAL_FALSE;
    } else
#endif
      RETVAL_STRING(buff);
  } else {
    RETVAL_FALSE;
  }

  zend_restore_error_handling(&error_handling);
}
/* }}} */
#endif

/* {{{ proto SplFileObject SplFileInfo::openFile([string mode = 'r' [, bool use_include_path  [, resource context]]])
   Open the current file */
SPL_METHOD(SplFileInfo, openFile)
{
  spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());

  spl_filesystem_object_create_type(ZEND_NUM_ARGS(), intern, SPL_FS_FILE, NULL, return_value);
}
/* }}} */

/* {{{ proto void SplFileInfo::setFileClass([string class_name])
   Class to use in openFile() */
SPL_METHOD(SplFileInfo, setFileClass)
{
  spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());
  zend_class_entry *ce = spl_ce_SplFileObject;
  zend_error_handling error_handling;

  zend_replace_error_handling(EH_THROW, spl_ce_UnexpectedValueException, &error_handling);

  if (zend_parse_parameters(ZEND_NUM_ARGS(), "|C", &ce) == SUCCESS) {
    intern->file_class = ce;
  }

  zend_restore_error_handling(&error_handling);
}
/* }}} */

/* {{{ proto void SplFileInfo::setInfoClass([string class_name])
   Class to use in getFileInfo(), getPathInfo() */
SPL_METHOD(SplFileInfo, setInfoClass)
{
  spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());
  zend_class_entry *ce = spl_ce_SplFileInfo;
  zend_error_handling error_handling;

  zend_replace_error_handling(EH_THROW, spl_ce_UnexpectedValueException, &error_handling );

  if (zend_parse_parameters(ZEND_NUM_ARGS(), "|C", &ce) == SUCCESS) {
    intern->info_class = ce;
  }

  zend_restore_error_handling(&error_handling);
}
/* }}} */

/* {{{ proto SplFileInfo SplFileInfo::getFileInfo([string $class_name])
   Get/copy file info */
SPL_METHOD(SplFileInfo, getFileInfo)
{
  spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());
  zend_class_entry *ce = intern->info_class;
  zend_error_handling error_handling;

  zend_replace_error_handling(EH_THROW, spl_ce_UnexpectedValueException, &error_handling);

  if (zend_parse_parameters(ZEND_NUM_ARGS(), "|C", &ce) == SUCCESS) {
    spl_filesystem_object_create_type(ZEND_NUM_ARGS(), intern, SPL_FS_INFO, ce, return_value);
  }

  zend_restore_error_handling(&error_handling);
}
/* }}} */

/* {{{ proto SplFileInfo SplFileInfo::getPathInfo([string $class_name])
   Get/copy file info */
SPL_METHOD(SplFileInfo, getPathInfo)
{
  spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());
  zend_class_entry *ce = intern->info_class;
  zend_error_handling error_handling;

  zend_replace_error_handling(EH_THROW, spl_ce_UnexpectedValueException, &error_handling);

  if (zend_parse_parameters(ZEND_NUM_ARGS(), "|C", &ce) == SUCCESS) {
    size_t path_len;
    char *path = spl_filesystem_object_get_pathname(intern, &path_len);
    if (path) {
      char *dpath = estrndup(path, path_len);
      path_len = php_dirname(dpath, path_len);
      spl_filesystem_object_create_info(intern, dpath, path_len, 1, ce, return_value);
      efree(dpath);
    }
  }

  zend_restore_error_handling(&error_handling);
}
/* }}} */

/* {{{  proto void SplFileInfo::_bad_state_ex(void) */
SPL_METHOD(SplFileInfo, _bad_state_ex)
{
  zend_throw_exception_ex(spl_ce_LogicException, 0,
    "The parent constructor was not called: the object is in an "
    "invalid state ");
}
/* }}} */

/* {{{ proto void FilesystemIterator::__construct(string path [, int flags])
 Cronstructs a new dir iterator from a path. */
SPL_METHOD(FilesystemIterator, __construct)
{
  spl_filesystem_object_construct(INTERNAL_FUNCTION_PARAM_PASSTHRU, DIT_CTOR_FLAGS | SPL_FILE_DIR_SKIPDOTS);
}
/* }}} */

/* {{{ proto void FilesystemIterator::rewind()
   Rewind dir back to the start */
SPL_METHOD(FilesystemIterator, rewind)
{
  spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());
  int skip_dots = SPL_HAS_FLAG(intern->flags, SPL_FILE_DIR_SKIPDOTS);

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  intern->u.dir.index = 0;
  if (intern->u.dir.dirp) {
    php_stream_rewinddir(intern->u.dir.dirp);
  }
  do {
    spl_filesystem_dir_read(intern);
  } while (skip_dots && spl_filesystem_is_dot(intern->u.dir.entry.d_name));
}
/* }}} */

/* {{{ proto int FilesystemIterator::getFlags()
   Get handling flags */
SPL_METHOD(FilesystemIterator, getFlags)
{
  spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  RETURN_LONG(intern->flags & (SPL_FILE_DIR_KEY_MODE_MASK | SPL_FILE_DIR_CURRENT_MODE_MASK | SPL_FILE_DIR_OTHERS_MASK));
} /* }}} */

/* {{{ proto void FilesystemIterator::setFlags(long $flags)
   Set handling flags */
SPL_METHOD(FilesystemIterator, setFlags)
{
  spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());
  zend_long flags;

  if (zend_parse_parameters(ZEND_NUM_ARGS(), "l", &flags) == FAILURE) {
    return;
  }

  intern->flags &= ~(SPL_FILE_DIR_KEY_MODE_MASK|SPL_FILE_DIR_CURRENT_MODE_MASK|SPL_FILE_DIR_OTHERS_MASK);
  intern->flags |= ((SPL_FILE_DIR_KEY_MODE_MASK|SPL_FILE_DIR_CURRENT_MODE_MASK|SPL_FILE_DIR_OTHERS_MASK) & flags);
} /* }}} */

/* {{{ proto bool RecursiveDirectoryIterator::hasChildren([bool $allow_links = false])
   Returns whether current entry is a directory and not '.' or '..' */
SPL_METHOD(RecursiveDirectoryIterator, hasChildren)
{
  zend_bool allow_links = 0;
  spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());

  if (zend_parse_parameters(ZEND_NUM_ARGS(), "|b", &allow_links) == FAILURE) {
    return;
  }
  if (spl_filesystem_is_invalid_or_dot(intern->u.dir.entry.d_name)) {
    RETURN_FALSE;
  } else {
    spl_filesystem_object_get_file_name(intern);
    if (!allow_links && !(intern->flags & SPL_FILE_DIR_FOLLOW_SYMLINKS)) {
      php_stat(intern->file_name, intern->file_name_len, FS_IS_LINK, return_value);
      if (zend_is_true(return_value)) {
        RETURN_FALSE;
      }
    }
    php_stat(intern->file_name, intern->file_name_len, FS_IS_DIR, return_value);
    }
```

```c
}
/* }}} */

/* {{{ proto RecursiveDirectoryIterator DirectoryIterator::getChildren()
   Returns an iterator for the current entry if it is a directory */
SPL_METHOD(RecursiveDirectoryIterator, getChildren)
{
  zval zpath, zflags;
  spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());
  spl_filesystem_object *subdir;
  char slash = SPL_HAS_FLAG(intern->flags, SPL_FILE_DIR_UNIXPATHS) ? '/' : DEFAULT_SLASH;

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  spl_filesystem_object_get_file_name(intern);

  ZVAL_LONG(&zflags, intern->flags);
  ZVAL_STRINGL(&zpath, intern->file_name, intern->file_name_len);
  spl_instantiate_arg_ex2(Z_OBJCE_P(getThis()), return_value, &zpath, &zflags);
  zval_ptr_dtor(&zpath);
  zval_ptr_dtor(&zflags);

  subdir = Z_SPLFILESYSTEM_P(return_value);
  if (subdir) {
    if (intern->u.dir.sub_path && intern->u.dir.sub_path[0]) {
      subdir->u.dir.sub_path_len = spprintf(&subdir->u.dir.sub_path, 0, "%s%c%s", intern->u.dir.sub_path, slash, intern->u.dir.entry.d_name);
    } else {
      subdir->u.dir.sub_path_len = strlen(intern->u.dir.entry.d_name);
      subdir->u.dir.sub_path = estrndup(intern->u.dir.entry.d_name, subdir->u.dir.sub_path_len);
    }
    subdir->info_class = intern->info_class;
    subdir->file_class = intern->file_class;
    subdir->oth = intern->oth;
  }
}
/* }}} */

/* {{{ proto void RecursiveDirectoryIterator::getSubPath()
   Get sub path */
SPL_METHOD(RecursiveDirectoryIterator, getSubPath)
{
  spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  if (intern->u.dir.sub_path) {
    RETURN_STRINGL(intern->u.dir.sub_path, intern->u.dir.sub_path_len);
  } else {
    RETURN_EMPTY_STRING();
  }
}
/* }}} */

/* {{{ proto void RecursiveDirectoryIterator::getSubPathname()
   Get sub path and file name */
SPL_METHOD(RecursiveDirectoryIterator, getSubPathname)
{
  spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());
  char slash = SPL_HAS_FLAG(intern->flags, SPL_FILE_DIR_UNIXPATHS) ? '/' : DEFAULT_SLASH;

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  if (intern->u.dir.sub_path) {
    RETURN_NEW_STR(strpprintf(0, "%s%c%s", intern->u.dir.sub_path, slash, intern->u.dir.entry.d_name));
  } else {
    RETURN_STRING(intern->u.dir.entry.d_name);
  }
}
/* }}} */

/* {{{ proto int RecursiveDirectoryIterator::__construct(string path [, int flags])
 Cronstructs a new dir iterator from a path. */
SPL_METHOD(RecursiveDirectoryIterator, __construct)
{
  spl_filesystem_object_construct(INTERNAL_FUNCTION_PARAM_PASSTHRU, DIT_CTOR_FLAGS);
}
/* }}} */

#ifdef HAVE_GLOB
/* {{{ proto int GlobIterator::__construct(string path [, int flags])
 Cronstructs a new dir iterator from a glob expression (no glob:// needed). */
SPL_METHOD(GlobIterator, __construct)
{
  spl_filesystem_object_construct(INTERNAL_FUNCTION_PARAM_PASSTHRU, DIT_CTOR_FLAGS|DIT_CTOR_GLOB);
}
/* }}} */

/* {{{ proto int GlobIterator::cont()
   Return the number of directories and files found by globbing */
SPL_METHOD(GlobIterator, count)
{
  spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  if (intern->u.dir.dirp && php_stream_is(intern->u.dir.dirp ,&php_glob_stream_ops)) {
    RETURN_LONG(php_glob_stream_get_count(intern->u.dir.dirp, NULL));
  } else {
    /* should not happen */
    php_error_docref(NULL, E_ERROR, "GlobIterator lost glob state");
  }
}
/* }}} */
#endif /* HAVE_GLOB */

/* {{{ forward declarations to the iterator handlers */
static void spl_filesystem_dir_it_dtor(zend_object_iterator *iter);
static int spl_filesystem_dir_it_valid(zend_object_iterator *iter);
static zval *spl_filesystem_dir_it_current_data(zend_object_iterator *iter);
static void spl_filesystem_dir_it_current_key(zend_object_iterator *iter, zval *key);
static void spl_filesystem_dir_it_move_forward(zend_object_iterator *iter);
static void spl_filesystem_dir_it_rewind(zend_object_iterator *iter);

/* iterator handler table */
static const zend_object_iterator_funcs spl_filesystem_dir_it_funcs = {
  spl_filesystem_dir_it_dtor,
  spl_filesystem_dir_it_valid,
  spl_filesystem_dir_it_current_data,
  spl_filesystem_dir_it_current_key,
  spl_filesystem_dir_it_move_forward,
  spl_filesystem_dir_it_rewind,
  NULL
};
/* }}} */

/* {{{ spl_ce_dir_get_iterator */
zend_object_iterator *spl_filesystem_dir_get_iterator(zend_class_entry *ce, zval *object, int by_ref)
{
  spl_filesystem_iterator *iterator;
  spl_filesystem_object *dir_object;

  if (by_ref) {
    zend_throw_exception(spl_ce_RuntimeException, "An iterator cannot be used with foreach by reference", 0);
    return NULL;
  }
  dir_object = Z_SPLFILESYSTEM_P(object);
  iterator = spl_filesystem_object_to_iterator(dir_object);
  ZVAL_COPY(&iterator->intern.data, object);
  iterator->intern.funcs = &spl_filesystem_dir_it_funcs;
  /* ->current must be initialized; rewind doesn't set it and valid
   * doesn't check whether it's set */
  iterator->current = *object;

  return &iterator->intern;
}
/* }}} */

/* {{{ spl_filesystem_dir_it_dtor */
static void spl_filesystem_dir_it_dtor(zend_object_iterator *iter)
{
  spl_filesystem_iterator *iterator = (spl_filesystem_iterator *)iter;

  if (!Z_ISUNDEF(iterator->intern.data)) {
    zval *object = &iterator->intern.data;
    zval_ptr_dtor(object);
  }
  /* Otherwise we were called from the owning object free storage handler as
   * it sets iterator->intern.data to IS_UNDEF.
   * We don't even need to destroy iterator->current as we didn't add a
   * reference to it in move_forward or get_iterator */
}
/* }}} */

/* {{{ spl_filesystem_dir_it_valid */
static int spl_filesystem_dir_it_valid(zend_object_iterator *iter)
{
  spl_filesystem_object *object = spl_filesystem_iterator_to_object((spl_filesystem_iterator *)iter);

  return object->u.dir.entry.d_name[0] != '\0' ? SUCCESS : FAILURE;
}
/* }}} */

/* {{{ spl_filesystem_dir_it_current_data */
static zval *spl_filesystem_dir_it_current_data(zend_object_iterator *iter)
{
  spl_filesystem_iterator *iterator = (spl_filesystem_iterator *)iter;

  return &iterator->current;
}
/* }}} */

/* {{{ spl_filesystem_dir_it_current_key */
static void spl_filesystem_dir_it_current_key(zend_object_iterator *iter, zval *key)
{
  spl_filesystem_object *object = spl_filesystem_iterator_to_object((spl_filesystem_iterator *)iter);

  ZVAL_LONG(key, object->u.dir.index);
}
/* }}} */

/* {{{ spl_filesystem_dir_it_move_forward */
static void spl_filesystem_dir_it_move_forward(zend_object_iterator *iter)
{
  spl_filesystem_object *object = spl_filesystem_iterator_to_object((spl_filesystem_iterator *)iter);

  object->u.dir.index++;
  spl_filesystem_dir_read(object);
  if (object->file_name) {
    efree(object->file_name);
    object->file_name = NULL;
  }
}
/* }}} */

/* {{{ spl_filesystem_dir_it_rewind */
static void spl_filesystem_dir_it_rewind(zend_object_iterator *iter)
{
  spl_filesystem_object *object = spl_filesystem_iterator_to_object((spl_filesystem_iterator *)iter);

  object->u.dir.index = 0;
  if (object->u.dir.dirp) {
    php_stream_rewinddir(object->u.dir.dirp);
  }
  spl_filesystem_dir_read(object);
}
/* }}} */

/* {{{ spl_filesystem_tree_it_dtor */
static void spl_filesystem_tree_it_dtor(zend_object_iterator *iter)
{
  spl_filesystem_iterator *iterator = (spl_filesystem_iterator *)iter;

  if (!Z_ISUNDEF(iterator->intern.data)) {
    zval *object = &iterator->intern.data;
    zval_ptr_dtor(object);
  } else {
    if (!Z_ISUNDEF(iterator->current)) {
      zval_ptr_dtor(&iterator->current);
      ZVAL_UNDEF(&iterator->current);
    }
  }
}
/* }}} */

/* {{{ spl_filesystem_tree_it_current_data */
static zval *spl_filesystem_tree_it_current_data(zend_object_iterator *iter)
{
  spl_filesystem_iterator *iterator = (spl_filesystem_iterator *)iter;
  spl_filesystem_object   *object   = spl_filesystem_iterator_to_object(iterator);

  if (SPL_FILE_DIR_CURRENT(object, SPL_FILE_DIR_CURRENT_AS_PATHNAME)) {
    if (Z_ISUNDEF(iterator->current)) {
      spl_filesystem_object_get_file_name(object);
      ZVAL_STRINGL(&iterator->current, object->file_name, object->file_name_len);
    }
    return &iterator->current;
  } else if (SPL_FILE_DIR_CURRENT(object, SPL_FILE_DIR_CURRENT_AS_FILEINFO)) {
    if (Z_ISUNDEF(iterator->current)) {
      spl_filesystem_object_get_file_name(object);
      spl_filesystem_object_create_type(0, object, SPL_FS_INFO, NULL, &iterator->current);
    }
    return &iterator->current;
  } else {
    return &iterator->intern.data;
  }
}
/* }}} */

/* {{{ spl_filesystem_tree_it_current_key */
static void spl_filesystem_tree_it_current_key(zend_object_iterator *iter, zval *key)
{
  spl_filesystem_object *object = spl_filesystem_iterator_to_object((spl_filesystem_iterator *)iter);

  if (SPL_FILE_DIR_KEY(object, SPL_FILE_DIR_KEY_AS_FILENAME)) {
    ZVAL_STRING(key, object->u.dir.entry.d_name);
  } else {
    spl_filesystem_object_get_file_name(object);
    ZVAL_STRINGL(key, object->file_name, object->file_name_len);
  }
}
/* }}} */

/* {{{ spl_filesystem_tree_it_move_forward */
static void spl_filesystem_tree_it_move_forward(zend_object_iterator *iter)
{
  spl_filesystem_iterator *iterator = (spl_filesystem_iterator *)iter;
  spl_filesystem_object   *object   = spl_filesystem_iterator_to_object(iterator);

  object->u.dir.index++;
  do {
    spl_filesystem_dir_read(object);
  } while (spl_filesystem_is_dot(object->u.dir.entry.d_name));
  if (object->file_name) {
    efree(object->file_name);
    object->file_name = NULL;
  }
  if (!Z_ISUNDEF(iterator->current)) {
    zval_ptr_dtor(&iterator->current);
    ZVAL_UNDEF(&iterator->current);
  }
}
/* }}} */

/* {{{ spl_filesystem_tree_it_rewind */
static void spl_filesystem_tree_it_rewind(zend_object_iterator *iter)
{
  spl_filesystem_iterator *iterator = (spl_filesystem_iterator *)iter;
  spl_filesystem_object   *object   = spl_filesystem_iterator_to_object(iterator);

  object->u.dir.index = 0;
  if (object->u.dir.dirp) {
    php_stream_rewinddir(object->u.dir.dirp);
  }
  do {
    spl_filesystem_dir_read(object);
  } while (spl_filesystem_is_dot(object->u.dir.entry.d_name));
  if (!Z_ISUNDEF(iterator->current)) {
    zval_ptr_dtor(&iterator->current);
    ZVAL_UNDEF(&iterator->current);
  }
}
/* }}} */

/* {{{ iterator handler table */
static const zend_object_iterator_funcs spl_filesystem_tree_it_funcs = {
  spl_filesystem_tree_it_dtor,
  spl_filesystem_dir_it_valid,
  spl_filesystem_tree_it_current_data,
  spl_filesystem_tree_it_current_key,
  spl_filesystem_tree_it_move_forward,
  spl_filesystem_tree_it_rewind,
  NULL
};
/* }}} */

/* {{{ spl_ce_dir_get_iterator */
zend_object_iterator *spl_filesystem_tree_get_iterator(zend_class_entry *ce, zval *object, int by_ref)
{
  spl_filesystem_iterator *iterator;
  spl_filesystem_object *dir_object;

  if (by_ref) {
    zend_throw_exception(spl_ce_RuntimeException, "An iterator cannot be used with foreach by reference", 0);
    return NULL;
  }
  dir_object = Z_SPLFILESYSTEM_P(object);
  iterator = spl_filesystem_object_to_iterator(dir_object);

  ZVAL_COPY(&iterator->intern.data, object);
  iterator->intern.funcs = &spl_filesystem_tree_it_funcs;

  return &iterator->intern;
}
/* }}} */

/* {{{ spl_filesystem_object_cast */
static int spl_filesystem_object_cast(zval *readobj, zval *writeobj, int type)
{
  spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(readobj);

  if (type == IS_STRING) {
    if (Z_OBJCE_P(readobj)->__tostring) {
      return std_object_handlers.cast_object(readobj, writeobj, type);
    }

    switch (intern->type) {
    case SPL_FS_INFO:
    case SPL_FS_FILE:
      ZVAL_STRINGL(writeobj, intern->file_name, intern->file_name_len);
      return SUCCESS;
    case SPL_FS_DIR:
      ZVAL_STRING(writeobj, intern->u.dir.entry.d_name);
      return SUCCESS;
    }
  } else if (type == _IS_BOOL) {
    ZVAL_TRUE(writeobj);
    return SUCCESS;
  }
  ZVAL_NULL(writeobj);
  return FAILURE;
}
/* }}} */

/* {{{ declare method parameters */
/* supply a name and default to call by parameter */
ZEND_BEGIN_ARG_INFO(arginfo_info___construct, 0)
  ZEND_ARG_INFO(0, file_name)
ZEND_END_ARG_INFO()

ZEND_BEGIN_ARG_INFO_EX(arginfo_info_openFile, 0, 0, 0)
  ZEND_ARG_INFO(0, open_mode)
  ZEND_ARG_INFO(0, use_include_path)
  ZEND_ARG_INFO(0, context)
ZEND_END_ARG_INFO()

ZEND_BEGIN_ARG_INFO_EX(arginfo_info_optinalFileClass, 0, 0, 0)
  ZEND_ARG_INFO(0, class_name)
ZEND_END_ARG_INFO()

ZEND_BEGIN_ARG_INFO_EX(arginfo_optinalSuffix, 0, 0, 0)
  ZEND_ARG_INFO(0, suffix)
ZEND_END_ARG_INFO()

ZEND_BEGIN_ARG_INFO(arginfo_splfileinfo_void, 0)
ZEND_END_ARG_INFO()

/* the method table */
/* each method can have its own parameters and visibility */
static const zend_function_entry spl_SplFileInfo_functions[] = {
  SPL_ME(SplFileInfo,       __construct,   arginfo_info___construct, ZEND_ACC_PUBLIC)
  SPL_ME(SplFileInfo,       getPath,       arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
  SPL_ME(SplFileInfo,       getFilename,   arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
  SPL_ME(SplFileInfo,       getExtension,  arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
  SPL_ME(SplFileInfo,       getBasename,   arginfo_optinalSuffix, ZEND_ACC_PUBLIC)
  SPL_ME(SplFileInfo,       getPathname,   arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
  SPL_ME(SplFileInfo,       getPerms,      arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
  SPL_ME(SplFileInfo,       getInode,      arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
  SPL_ME(SplFileInfo,       getSize,       arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
  SPL_ME(SplFileInfo,       getOwner,      arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
  SPL_ME(SplFileInfo,       getGroup,      arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
  SPL_ME(SplFileInfo,       getATime,      arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
  SPL_ME(SplFileInfo,       getMTime,      arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
  SPL_ME(SplFileInfo,       getCTime,      arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
  SPL_ME(SplFileInfo,       getType,       arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
  SPL_ME(SplFileInfo,       isWritable,    arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
  SPL_ME(SplFileInfo,       isReadable,    arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
  SPL_ME(SplFileInfo,       isExecutable,  arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
  SPL_ME(SplFileInfo,       isFile,        arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
  SPL_ME(SplFileInfo,       isDir,         arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
  SPL_ME(SplFileInfo,       isLink,        arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
  SPL_ME(SplFileInfo,       getLinkTarget, arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
#if HAVE_REALPATH || defined(ZTS)
  SPL_ME(SplFileInfo,       getRealPath,   arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
#endif
  SPL_ME(SplFileInfo,       getFileInfo,   arginfo_info_optinalFileClass, ZEND_ACC_PUBLIC)
  SPL_ME(SplFileInfo,       getPathInfo,   arginfo_info_optinalFileClass, ZEND_ACC_PUBLIC)
  SPL_ME(SplFileInfo,       openFile,      arginfo_info_openFile,         ZEND_ACC_PUBLIC)
  SPL_ME(SplFileInfo,       setFileClass,  arginfo_info_optinalFileClass, ZEND_ACC_PUBLIC)
  SPL_ME(SplFileInfo,       setInfoClass,  arginfo_info_optinalFileClass, ZEND_ACC_PUBLIC)
  SPL_ME(SplFileInfo,       _bad_state_ex, NULL,              ZEND_ACC_PUBLIC|ZEND_ACC_FINAL)
  SPL_MA(SplFileInfo,       __toString, SplFileInfo, getPathname, arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
  PHP_FE_END
};

ZEND_BEGIN_ARG_INFO(arginfo_dir___construct, 0)
  ZEND_ARG_INFO(0, path)
ZEND_END_ARG_INFO()

ZEND_BEGIN_ARG_INFO(arginfo_dir_it_seek, 0)
  ZEND_ARG_INFO(0, position)
ZEND_END_ARG_INFO();

/* the method table */
/* each method can have its own parameters and visibility */
static const zend_function_entry spl_DirectoryIterator_functions[] = {
  SPL_ME(DirectoryIterator, __construct,   arginfo_dir___construct, ZEND_ACC_PUBLIC)
  SPL_ME(DirectoryIterator, getFilename,   arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
  SPL_ME(DirectoryIterator, getExtension,  arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
  SPL_ME(DirectoryIterator, getBasename,   arginfo_optinalSuffix, ZEND_ACC_PUBLIC)
  SPL_ME(DirectoryIterator, isDot,         arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
  SPL_ME(DirectoryIterator, rewind,        arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
  SPL_ME(DirectoryIterator, valid,         arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
  SPL_ME(DirectoryIterator, key,           arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
  SPL_ME(DirectoryIterator, current,       arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
  SPL_ME(DirectoryIterator, next,          arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
  SPL_ME(DirectoryIterator, seek,          arginfo_dir_it_seek, ZEND_ACC_PUBLIC)
  SPL_MA(DirectoryIterator, __toString, DirectoryIterator, getFilename, arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
  PHP_FE_END
};

ZEND_BEGIN_ARG_INFO_EX(arginfo_r_dir___construct, 0, 0, 1)
  ZEND_ARG_INFO(0, path)
  ZEND_ARG_INFO(0, flags)
ZEND_END_ARG_INFO()

ZEND_BEGIN_ARG_INFO_EX(arginfo_r_dir_hasChildren, 0, 0, 0)
  ZEND_ARG_INFO(0, allow_links)
ZEND_END_ARG_INFO()

ZEND_BEGIN_ARG_INFO_EX(arginfo_r_dir_setFlags, 0, 0, 0)
  ZEND_ARG_INFO(0, flags)
ZEND_END_ARG_INFO()

static const zend_function_entry spl_FilesystemIterator_functions[] = {
  SPL_ME(FilesystemIterator, __construct,   arginfo_r_dir___construct, ZEND_ACC_PUBLIC)
  SPL_ME(FilesystemIterator, rewind,        arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
  SPL_ME(DirectoryIterator,  next,          arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
  SPL_ME(FilesystemIterator, key,           arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
  SPL_ME(FilesystemIterator, current,       arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
  SPL_ME(FilesystemIterator, getFlags,      arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
  SPL_ME(FilesystemIterator, setFlags,      arginfo_r_dir_setFlags, ZEND_ACC_PUBLIC)
  PHP_FE_END
};

static const zend_function_entry spl_RecursiveDirectoryIterator_functions[] = {
  SPL_ME(RecursiveDirectoryIterator, __construct,   arginfo_r_dir___construct, ZEND_ACC_PUBLIC)
  SPL_ME(RecursiveDirectoryIterator, hasChildren,   arginfo_r_dir_hasChildren, ZEND_ACC_PUBLIC)
  SPL_ME(RecursiveDirectoryIterator, getChildren,   arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
  SPL_ME(RecursiveDirectoryIterator, getSubPath,    arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
  SPL_ME(RecursiveDirectoryIterator, getSubPathname,arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
  PHP_FE_END
};

#ifdef HAVE_GLOB
static const zend_function_entry spl_GlobIterator_functions[] = {
  SPL_ME(GlobIterator, __construct,   arginfo_r_dir___construct, ZEND_ACC_PUBLIC)
  SPL_ME(GlobIterator, count,         arginfo_splfileinfo_void,  ZEND_ACC_PUBLIC)
  PHP_FE_END
};
#endif
/* }}} */

static int spl_filesystem_file_read(spl_filesystem_object *intern, int silent) /* {{{ */
{
  char *buf;
  size_t line_len = 0;
  zend_long line_add = (intern->u.file.current_line || !Z_ISUNDEF(intern->u.file.current_zval)) ? 1 : 0;

  spl_filesystem_file_free_line(intern);

  if (php_stream_eof(intern->u.file.stream)) {
    if (!silent) {
      zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Cannot read from file %s", intern->file_name);
    }
    return FAILURE;
  }

  if (intern->u.file.max_line_len > 0) {
    buf = safe_emalloc((intern->u.file.max_line_len + 1), sizeof(char), 0);
    if (php_stream_get_line(intern->u.file.stream, buf, intern->u.file.max_line_len + 1, &line_len) == NULL) {
      efree(buf);
      buf = NULL;
    } else {
      buf[line_len] = '\0';
    }
  } else {
    buf = php_stream_get_line(intern->u.file.stream, NULL, 0, &line_len);
  }

  if (!buf) {
    intern->u.file.current_line = estrdup("");
    intern->u.file.current_line_len = 0;
  } else {
    if (SPL_HAS_FLAG(intern->flags, SPL_FILE_OBJECT_DROP_NEW_LINE)) {
      line_len = strcspn(buf, "\r\n");
      buf[line_len] = '\0';
    }

    intern->u.file.current_line = buf;
    intern->u.file.current_line_len = line_len;
  }
  intern->u.file.current_line_num += line_add;

  return SUCCESS;
} /* }}} */

static int spl_filesystem_file_call(spl_filesystem_object *intern, zend_function *func_ptr, int pass_num_args, zval *return_value, zval *arg2) /* {{{ */
{
  zend_fcall_info fci;
  zend_fcall_info_cache fcic;
  zval *zresource_ptr = &intern->u.file.zresource, retval;
  int result;
  int num_args = pass_num_args + (arg2 ? 2 : 1);

  zval *params = (zval*)safe_emalloc(num_args, sizeof(zval), 0);

  params[0] = *zresource_ptr;

  if (arg2) {
    params[1] = *arg2;
  }

  if (zend_get_parameters_array_ex(pass_num_args, params + (arg2 ? 2 : 1)) != SUCCESS) {
    efree(params);
    WRONG_PARAM_COUNT_WITH_RETVAL(FAILURE);
  }

  ZVAL_UNDEF(&retval);

  fci.size = sizeof(fci);
  fci.object = NULL;
  fci.retval = &retval;
  fci.param_count = num_args;
  fci.params = params;
  fci.no_separation = 1;
  ZVAL_STR(&fci.function_name, func_ptr->common.function_name);

  fcic.function_handler = func_ptr;
  fcic.calling_scope = NULL;
  fcic.called_scope = NULL;
  fcic.object = NULL;

  result = zend_call_function(&fci, &fcic);

  if (result == FAILURE || Z_ISUNDEF(retval)) {
    RETVAL_FALSE;
  } else {
    ZVAL_ZVAL(return_value, &retval, 0, 0);
  }

  efree(params);
  return result;
} /* }}} */

#define FileFunctionCall(func_name, pass_num_args, arg2) /* {{{ */ \
{ \
  zend_function *func_ptr; \
  func_ptr = (zend_function *)zend_hash_str_find_ptr(EG(function_table), #func_name, sizeof(#func_name) - 1); \
  if (func_ptr == NULL) { \
    zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Internal error, function '%s' not found. Please report", #func_name); \
    return; \
  } \
  spl_filesystem_file_call(intern, func_ptr, pass_num_args, return_value, arg2); \
} /* }}} */

static int spl_filesystem_file_read_csv(spl_filesystem_object *intern, char delimiter, char enclosure, char escape, zval *return_value) /* {{{ */
{
  int ret = SUCCESS;
  zval *value;

  do {
    ret = spl_filesystem_file_read(intern, 1);
  } while (ret == SUCCESS && !intern->u.file.current_line_len && SPL_HAS_FLAG(intern->flags, SPL_FILE_OBJECT_SKIP_EMPTY));

  if (ret == SUCCESS) {
    size_t buf_len = intern->u.file.current_line_len;
    char *buf = estrndup(intern->u.file.current_line, buf_len);

    if (!Z_ISUNDEF(intern->u.file.current_zval)) {
      zval_ptr_dtor(&intern->u.file.current_zval);
      ZVAL_UNDEF(&intern->u.file.current_zval);
    }

    php_fgetcsv(intern->u.file.stream, delimiter, enclosure, escape, buf_len, buf, &intern->u.file.current_zval);
    if (return_value) {
      zval_ptr_dtor(return_value);
      value = &intern->u.file.current_zval;
      ZVAL_DEREF(value);
      ZVAL_COPY(return_value, value);
    }
  }
  return ret;
}
/* }}} */

static int spl_filesystem_file_read_line_ex(zval * this_ptr, spl_filesystem_object *intern, int silent) /* {{{ */
{
  zval retval;

  /* 1) use fgetcsv? 2) overloaded call the function, 3) do it directly */
  if (SPL_HAS_FLAG(intern->flags, SPL_FILE_OBJECT_READ_CSV) || intern->u.file.func_getCurr->common.scope != spl_ce_SplFileObject) {
    if (php_stream_eof(intern->u.file.stream)) {
      if (!silent) {
        zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Cannot read from file %s", intern->file_name);
      }
      return FAILURE;
    }
    if (SPL_HAS_FLAG(intern->flags, SPL_FILE_OBJECT_READ_CSV)) {
      return spl_filesystem_file_read_csv(intern, intern->u.file.delimiter, intern->u.file.enclosure, intern->u.file.escape, NULL);
    } else {
      zend_execute_data *execute_data = EG(current_execute_data);
      zend_call_method_with_0_params(this_ptr, Z_OBJCE(EX(This)), &intern->u.file.func_getCurr, "getCurrentLine", &retval);
    }
    if (!Z_ISUNDEF(retval)) {
      if (intern->u.file.current_line || !Z_ISUNDEF(intern->u.file.current_zval)) {
        intern->u.file.current_line_num++;
      }
      spl_filesystem_file_free_line(intern);
      if (Z_TYPE(retval) == IS_STRING) {
        intern->u.file.current_line = estrndup(Z_STRVAL(retval), Z_STRLEN(retval));
        intern->u.file.current_line_len = Z_STRLEN(retval);
      } else {
        zval *value = &retval;

        ZVAL_DEREF(value);
        ZVAL_COPY(&intern->u.file.current_zval, value);
      }
      zval_ptr_dtor(&retval);
      return SUCCESS;
    } else {
      return FAILURE;
    }
  } else {
    return spl_filesystem_file_read(intern, silent);
  }
} /* }}} */

static int spl_filesystem_file_is_empty_line(spl_filesystem_object *intern) /* {{{ */
{
  if (intern->u.file.current_line) {
    return intern->u.file.current_line_len == 0;
  } else if (!Z_ISUNDEF(intern->u.file.current_zval)) {
    switch(Z_TYPE(intern->u.file.current_zval)) {
      case IS_STRING:
        return Z_STRLEN(intern->u.file.current_zval) == 0;
      case IS_ARRAY:
        if (SPL_HAS_FLAG(intern->flags, SPL_FILE_OBJECT_READ_CSV)
            && zend_hash_num_elements(Z_ARRVAL(intern->u.file.current_zval)) == 1) {
          uint32_t idx = 0;
          zval *first;

          while (Z_ISUNDEF(Z_ARRVAL(intern->u.file.current_zval)->arData[idx].val)) {
            idx++;
          }
          first = &Z_ARRVAL(intern->u.file.current_zval)->arData[idx].val;
          return Z_TYPE_P(first) == IS_STRING && Z_STRLEN_P(first) == 0;
        }
        return zend_hash_num_elements(Z_ARRVAL(intern->u.file.current_zval)) == 0;
      case IS_NULL:
        return 1;
      default:
        return 0;
    }
  } else {
    return 1;
  }
}
/* }}} */

static int spl_filesystem_file_read_line(zval * this_ptr, spl_filesystem_object *intern, int silent) /* {{{ */
{
  int ret = spl_filesystem_file_read_line_ex(this_ptr, intern, silent);

  while (SPL_HAS_FLAG(intern->flags, SPL_FILE_OBJECT_SKIP_EMPTY) && ret == SUCCESS && spl_filesystem_file_is_empty_line(intern)) {
    spl_filesystem_file_free_line(intern);
    ret = spl_filesystem_file_read_line_ex(this_ptr, intern, silent);
  }

  return ret;
}
/* }}} */

static void spl_filesystem_file_rewind(zval * this_ptr, spl_filesystem_object *intern) /* {{{ */
{
  if(!intern->u.file.stream) {
    zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Object not initialized");
    return;
  }
  if (-1 == php_stream_rewind(intern->u.file.stream)) {
    zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Cannot rewind file %s", intern->file_name);
  } else {
    spl_filesystem_file_free_line(intern);
    intern->u.file.current_line_num = 0;
  }
  if (SPL_HAS_FLAG(intern->flags, SPL_FILE_OBJECT_READ_AHEAD)) {
    spl_filesystem_file_read_line(this_ptr, intern, 1);
  }
} /* }}} */

/* {{{ proto void SplFileObject::__construct(string filename [, string mode = 'r' [, bool use_include_path  [, resource context]]]])
   Construct a new file object */
SPL_METHOD(SplFileObject, __construct)
{
  spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());
  zend_bool use_include_path = 0;
  char *p1, *p2;
  char *tmp_path;
  size_t   tmp_path_len;
  zend_error_handling error_handling;

  intern->u.file.open_mode = NULL;
  intern->u.file.open_mode_len = 0;

  if (zend_parse_parameters_throw(ZEND_NUM_ARGS(), "p|sbr!",
      &intern->file_name, &intern->file_name_len,
      &intern->u.file.open_mode, &intern->u.file.open_mode_len,
      &use_include_path, &intern->u.file.zcontext) == FAILURE) {
    intern->u.file.open_mode = NULL;
    intern->file_name = NULL;
    return;
  }

  if (intern->u.file.open_mode == NULL) {
    intern->u.file.open_mode = "r";
    intern->u.file.open_mode_len = 1;
  }

  zend_replace_error_handling(EH_THROW, spl_ce_RuntimeException, &error_handling);

  if (spl_filesystem_file_open(intern, use_include_path, 0) == SUCCESS) {
    tmp_path_len = strlen(intern->u.file.stream->orig_path);

    if (tmp_path_len > 1 && IS_SLASH_AT(intern->u.file.stream->orig_path, tmp_path_len-1)) {
      tmp_path_len--;
    }

    tmp_path = estrndup(intern->u.file.stream->orig_path, tmp_path_len);

    p1 = strrchr(tmp_path, '/');
#if defined(PHP_WIN32)
    p2 = strrchr(tmp_path, '\\');
#else
    p2 = 0;
#endif
    if (p1 || p2) {
      intern->_path_len = ((p1 > p2 ? p1 : p2) - tmp_path);
    } else {
      intern->_path_len = 0;
    }

    efree(tmp_path);

    intern->_path = estrndup(intern->u.file.stream->orig_path, intern->_path_len);
  }

  zend_restore_error_handling(&error_handling);

} /* }}} */

/* {{{ proto void SplTempFileObject::__construct([int max_memory])
   Construct a new temp file object */
SPL_METHOD(SplTempFileObject, __construct)
{
  zend_long max_memory = PHP_STREAM_MAX_MEM;
  char tmp_fname[48];
  spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());
  zend_error_handling error_handling;

  if (zend_parse_parameters_throw(ZEND_NUM_ARGS(), "|l", &max_memory) == FAILURE) {
    return;
  }

  if (max_memory < 0) {
    intern->file_name = "php://memory";
    intern->file_name_len = 12;
  } else if (ZEND_NUM_ARGS()) {
    intern->file_name_len = slprintf(tmp_fname, sizeof(tmp_fname), "php://temp/maxmemory:" ZEND_LONG_FMT, max_memory);
    intern->file_name = tmp_fname;
  } else {
    intern->file_name = "php://temp";
    intern->file_name_len = 10;
  }
  intern->u.file.open_mode = "wb";
  intern->u.file.open_mode_len = 1;

  zend_replace_error_handling(EH_THROW, spl_ce_RuntimeException, &error_handling);
  if (spl_filesystem_file_open(intern, 0, 0) == SUCCESS) {
    intern->_path_len = 0;
    intern->_path = estrndup("", 0);
  }
  zend_restore_error_handling(&error_handling);
} /* }}} */

/* {{{ proto void SplFileObject::rewind()
   Rewind the file and read the first line */
SPL_METHOD(SplFileObject, rewind)
{
  spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  spl_filesystem_file_rewind(getThis(), intern);
} /* }}} */

/* {{{ proto void SplFileObject::eof()
   Return whether end of file is reached */
SPL_METHOD(SplFileObject, eof)
{
  spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  if(!intern->u.file.stream) {
    zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Object not initialized");
    return;
  }

  RETURN_BOOL(php_stream_eof(intern->u.file.stream));
} /* }}} */

/* {{{ proto void SplFileObject::valid()
   Return !eof() */
SPL_METHOD(SplFileObject, valid)
{
  spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  if (SPL_HAS_FLAG(intern->flags, SPL_FILE_OBJECT_READ_AHEAD)) {
    RETURN_BOOL(intern->u.file.current_line || !Z_ISUNDEF(intern->u.file.current_zval));
  } else {
    if(!intern->u.file.stream) {
      RETURN_FALSE;
    }
    RETVAL_BOOL(!php_stream_eof(intern->u.file.stream));
  }
} /* }}} */

/* {{{ proto string SplFileObject::fgets()
   Rturn next line from file */
SPL_METHOD(SplFileObject, fgets)
{
  spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  if(!intern->u.file.stream) {
    zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Object not initialized");
    return;
  }

  if (spl_filesystem_file_read(intern, 0) == FAILURE) {
    RETURN_FALSE;
  }
  RETURN_STRINGL(intern->u.file.current_line, intern->u.file.current_line_len);
} /* }}} */

/* {{{ proto string SplFileObject::current()
   Return current line from file */
SPL_METHOD(SplFileObject, current)
{
  spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  if(!intern->u.file.stream) {
    zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Object not initialized");
    return;
  }

  if (!intern->u.file.current_line && Z_ISUNDEF(intern->u.file.current_zval)) {
    spl_filesystem_file_read_line(getThis(), intern, 1);
  }
  if (intern->u.file.current_line && (!SPL_HAS_FLAG(intern->flags, SPL_FILE_OBJECT_READ_CSV) || Z_ISUNDEF(intern->u.file.current_zval))) {
    RETURN_STRINGL(intern->u.file.current_line, intern->u.file.current_line_len);
  } else if (!Z_ISUNDEF(intern->u.file.current_zval)) {
    zval *value = &intern->u.file.current_zval;

    ZVAL_DEREF(value);
    ZVAL_COPY(return_value, value);
    return;
  }
  RETURN_FALSE;
} /* }}} */

/* {{{ proto int SplFileObject::key()
   Return line number */
SPL_METHOD(SplFileObject, key)
{
  spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

/*  Do not read the next line to support correct counting with fgetc()
  if (!intern->current_line) {
    spl_filesystem_file_read_line(getThis(), intern, 1);
  } */
  RETURN_LONG(intern->u.file.current_line_num);
} /* }}} */

/* {{{ proto void SplFileObject::next()
   Read next line */
SPL_METHOD(SplFileObject, next)
{
  spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  spl_filesystem_file_free_line(intern);
  if (SPL_HAS_FLAG(intern->flags, SPL_FILE_OBJECT_READ_AHEAD)) {
    spl_filesystem_file_read_line(getThis(), intern, 1);
  }
  intern->u.file.current_line_num++;
} /* }}} */

/* {{{ proto void SplFileObject::setFlags(int flags)
   Set file handling flags */
SPL_METHOD(SplFileObject, setFlags)
{
  spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());

  if (zend_parse_parameters(ZEND_NUM_ARGS(), "l", &intern->flags) == FAILURE) {
    return;
  }
} /* }}} */

/* {{{ proto int SplFileObject::getFlags()
   Get file handling flags */
SPL_METHOD(SplFileObject, getFlags)
{
  spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  RETURN_LONG(intern->flags & SPL_FILE_OBJECT_MASK);
} /* }}} */

/* {{{ proto void SplFileObject::setMaxLineLen(int max_len)
   Set maximum line length */
SPL_METHOD(SplFileObject, setMaxLineLen)
{
  zend_long max_len;

  spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());

  if (zend_parse_parameters(ZEND_NUM_ARGS(), "l", &max_len) == FAILURE) {
    return;
  }

  if (max_len < 0) {
    zend_throw_exception_ex(spl_ce_DomainException, 0, "Maximum line length must be greater than or equal zero");
    return;
  }

  intern->u.file.max_line_len = max_len;
} /* }}} */

/* {{{ proto int SplFileObject::getMaxLineLen()
   Get maximum line length */
SPL_METHOD(SplFileObject, getMaxLineLen)
{
  spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  RETURN_LONG((zend_long)intern->u.file.max_line_len);
} /* }}} */

/* {{{ proto bool SplFileObject::hasChildren()
   Return false */
SPL_METHOD(SplFileObject, hasChildren)
{
  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  RETURN_FALSE;
} /* }}} */

/* {{{ proto bool SplFileObject::getChildren()
   Read NULL */
SPL_METHOD(SplFileObject, getChildren)
{
  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }
  /* return NULL */
} /* }}} */

/* {{{ FileFunction */
#define FileFunction(func_name) \
SPL_METHOD(SplFileObject, func_name) \
{ \
  spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis()); \
  FileFunctionCall(func_name, ZEND_NUM_ARGS(), NULL); \
}
/* }}} */

/* {{{ proto array SplFileObject::fgetcsv([string delimiter [, string enclosure [, escape = '\\']]])
   Return current line as csv */
SPL_METHOD(SplFileObject, fgetcsv)
{
  spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());
  char delimiter = intern->u.file.delimiter, enclosure = intern->u.file.enclosure, escape = intern->u.file.escape;
  char *delim = NULL, *enclo = NULL, *esc = NULL;
  size_t d_len = 0, e_len = 0, esc_len = 0;

  if (zend_parse_parameters(ZEND_NUM_ARGS(), "|sss", &delim, &d_len, &enclo, &e_len, &esc, &esc_len) == SUCCESS) {

    if(!intern->u.file.stream) {
      zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Object not initialized");
      return;
    }

    switch(ZEND_NUM_ARGS())
    {
    case 3:
      if (esc_len != 1) {
        php_error_docref(NULL, E_WARNING, "escape must be a character");
        RETURN_FALSE;
      }
      escape = esc[0];
      /* no break */
    case 2:
      if (e_len != 1) {
        php_error_docref(NULL, E_WARNING, "enclosure must be a character");
        RETURN_FALSE;
      }
      enclosure = enclo[0];
      /* no break */
    case 1:
      if (d_len != 1) {
        php_error_docref(NULL, E_WARNING, "delimiter must be a character");
        RETURN_FALSE;
      }
      delimiter = delim[0];
      /* no break */
    case 0:
      break;
    }
    spl_filesystem_file_read_csv(intern, delimiter, enclosure, escape, return_value);
  }
}
/* }}} */

/* {{{ proto int SplFileObject::fputcsv(array fields, [string delimiter [, string enclosure [, string escape]]])
   Output a field array as a CSV line */
SPL_METHOD(SplFileObject, fputcsv)
{
  spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());
  char delimiter = intern->u.file.delimiter, enclosure = intern->u.file.enclosure, escape = intern->u.file.escape;
  char *delim = NULL, *enclo = NULL, *esc = NULL;
  size_t d_len = 0, e_len = 0, esc_len = 0;
  zend_long ret;
  zval *fields = NULL;

  if (zend_parse_parameters(ZEND_NUM_ARGS(), "a|sss", &fields, &delim, &d_len, &enclo, &e_len, &esc, &esc_len) == SUCCESS) {
    switch(ZEND_NUM_ARGS())
    {
    case 4:
      if (esc_len != 1) {
        php_error_docref(NULL, E_WARNING, "escape must be a character");
        RETURN_FALSE;
      }
      escape = esc[0];
      /* no break */
    case 3:
      if (e_len != 1) {
        php_error_docref(NULL, E_WARNING, "enclosure must be a character");
        RETURN_FALSE;
      }
      enclosure = enclo[0];
      /* no break */
    case 2:
      if (d_len != 1) {
        php_error_docref(NULL, E_WARNING, "delimiter must be a character");
        RETURN_FALSE;
      }
      delimiter = delim[0];
      /* no break */
    case 1:
    case 0:
      break;
    }
    ret = php_fputcsv(intern->u.file.stream, fields, delimiter, enclosure, escape);
    RETURN_LONG(ret);
  }
}
/* }}} */

/* {{{ proto void SplFileObject::setCsvControl([string delimiter [, string enclosure [, string escape ]]])
   Set the delimiter, enclosure and escape character used in fgetcsv */
SPL_METHOD(SplFileObject, setCsvControl)
{
  spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());
  char delimiter = ',', enclosure = '"', escape='\\';
  char *delim = NULL, *enclo = NULL, *esc = NULL;
  size_t d_len = 0, e_len = 0, esc_len = 0;

  if (zend_parse_parameters(ZEND_NUM_ARGS(), "|sss", &delim, &d_len, &enclo, &e_len, &esc, &esc_len) == SUCCESS) {
    switch(ZEND_NUM_ARGS())
    {
    case 3:
      if (esc_len != 1) {
        php_error_docref(NULL, E_WARNING, "escape must be a character");
        RETURN_FALSE;
      }
      escape = esc[0];
      /* no break */
    case 2:
      if (e_len != 1) {
        php_error_docref(NULL, E_WARNING, "enclosure must be a character");
        RETURN_FALSE;
      }
      enclosure = enclo[0];
      /* no break */
    case 1:
      if (d_len != 1) {
        php_error_docref(NULL, E_WARNING, "delimiter must be a character");
        RETURN_FALSE;
      }
      delimiter = delim[0];
      /* no break */
    case 0:
      break;
    }
    intern->u.file.delimiter = delimiter;
    intern->u.file.enclosure = enclosure;
    intern->u.file.escape    = escape;
  }
}
/* }}} */

/* {{{ proto array SplFileObject::getCsvControl()
   Get the delimiter, enclosure and escape character used in fgetcsv */
SPL_METHOD(SplFileObject, getCsvControl)
{
  spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());
  char delimiter[2], enclosure[2], escape[2];

  array_init(return_value);

  delimiter[0] = intern->u.file.delimiter;
  delimiter[1] = '\0';
  enclosure[0] = intern->u.file.enclosure;
  enclosure[1] = '\0';
  escape[0] = intern->u.file.escape;
  escape[1] = '\0';

  add_next_index_string(return_value, delimiter);
  add_next_index_string(return_value, enclosure);
  add_next_index_string(return_value, escape);
}
/* }}} */

/* {{{ proto bool SplFileObject::flock(int operation [, int &wouldblock])
   Portable file locking */
FileFunction(flock)
/* }}} */

/* {{{ proto bool SplFileObject::fflush()
   Flush the file */
SPL_METHOD(SplFileObject, fflush)
{
  spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());

  if(!intern->u.file.stream) {
    zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Object not initialized");
    return;
  }

  RETURN_BOOL(!php_stream_flush(intern->u.file.stream));
} /* }}} */

/* {{{ proto int SplFileObject::ftell()
   Return current file position */
SPL_METHOD(SplFileObject, ftell)
{
  spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());
  zend_long ret;

  if(!intern->u.file.stream) {
    zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Object not initialized");
    return;
  }

  ret = php_stream_tell(intern->u.file.stream);

  if (ret == -1) {
    RETURN_FALSE;
  } else {
    RETURN_LONG(ret);
  }
} /* }}} */

/* {{{ proto int SplFileObject::fseek(int pos [, int whence = SEEK_SET])
   Return current file position */
SPL_METHOD(SplFileObject, fseek)
{
  spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());
  zend_long pos, whence = SEEK_SET;

  if (zend_parse_parameters(ZEND_NUM_ARGS(), "l|l", &pos, &whence) == FAILURE) {
    return;
  }

  if(!intern->u.file.stream) {
    zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Object not initialized");
    return;
  }

  spl_filesystem_file_free_line(intern);
  RETURN_LONG(php_stream_seek(intern->u.file.stream, pos, (int)whence));
} /* }}} */

/* {{{ proto int SplFileObject::fgetc()
   Get a character form the file */
SPL_METHOD(SplFileObject, fgetc)
{
  spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());
  char buf[2];
  int result;

  if(!intern->u.file.stream) {
    zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Object not initialized");
    return;
  }

  spl_filesystem_file_free_line(intern);

  result = php_stream_getc(intern->u.file.stream);

  if (result == EOF) {
    RETVAL_FALSE;
  } else {
    if (result == '\n') {
      intern->u.file.current_line_num++;
    }
    buf[0] = result;
    buf[1] = '\0';

    RETURN_STRINGL(buf, 1);
  }
} /* }}} */

/* {{{ proto string SplFileObject::fgetss([string allowable_tags])
   Get a line from file pointer and strip HTML tags */
SPL_METHOD(SplFileObject, fgetss)
{
  spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());
  zval arg2;

  if(!intern->u.file.stream) {
    zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Object not initialized");
    return;
  }

  if (intern->u.file.max_line_len > 0) {
    ZVAL_LONG(&arg2, intern->u.file.max_line_len);
  } else {
    ZVAL_LONG(&arg2, 1024);
  }

  spl_filesystem_file_free_line(intern);
  intern->u.file.current_line_num++;

  FileFunctionCall(fgetss, ZEND_NUM_ARGS(), &arg2);
} /* }}} */

/* {{{ proto int SplFileObject::fpassthru()
   Output all remaining data from a file pointer */
SPL_METHOD(SplFileObject, fpassthru)
{
  spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());

  if(!intern->u.file.stream) {
    zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Object not initialized");
    return;
  }

  RETURN_LONG(php_stream_passthru(intern->u.file.stream));
} /* }}} */

/* {{{ proto bool SplFileObject::fscanf(string format [, string ...])
   Implements a mostly ANSI compatible fscanf() */
SPL_METHOD(SplFileObject, fscanf)
{
  spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());

  if(!intern->u.file.stream) {
    zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Object not initialized");
    return;
  }

  spl_filesystem_file_free_line(intern);
  intern->u.file.current_line_num++;

  FileFunctionCall(fscanf, ZEND_NUM_ARGS(), NULL);
}
/* }}} */

/* {{{ proto mixed SplFileObject::fwrite(string str [, int length])
   Binary-safe file write */
SPL_METHOD(SplFileObject, fwrite)
{
  spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());
  char *str;
  size_t str_len;
  zend_long length = 0;

  if (zend_parse_parameters(ZEND_NUM_ARGS(), "s|l", &str, &str_len, &length) == FAILURE) {
    return;
  }

  if(!intern->u.file.stream) {
    zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Object not initialized");
    return;
  }

  if (ZEND_NUM_ARGS() > 1) {
    if (length >= 0) {
      str_len = MIN((size_t)length, str_len);
    } else {
      /* Negative length given, nothing to write */
      str_len = 0;
    }
  }
  if (!str_len) {
    RETURN_LONG(0);
  }

  RETURN_LONG(php_stream_write(intern->u.file.stream, str, str_len));
} /* }}} */

SPL_METHOD(SplFileObject, fread)
{
  spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());
  zend_long length = 0;

  if (zend_parse_parameters(ZEND_NUM_ARGS(), "l", &length) == FAILURE) {
    return;
  }

  if(!intern->u.file.stream) {
    zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Object not initialized");
    return;
  }

  if (length <= 0) {
    php_error_docref(NULL, E_WARNING, "Length parameter must be greater than 0");
    RETURN_FALSE;
  }

  ZVAL_NEW_STR(return_value, zend_string_alloc(length, 0));
  Z_STRLEN_P(return_value) = php_stream_read(intern->u.file.stream, Z_STRVAL_P(return_value), length);

  /* needed because recv/read/gzread doesnt put a null at the end*/
  Z_STRVAL_P(return_value)[Z_STRLEN_P(return_value)] = 0;
}

/* {{{ proto bool SplFileObject::fstat()
   Stat() on a filehandle */
FileFunction(fstat)
/* }}} */

/* {{{ proto bool SplFileObject::ftruncate(int size)
   Truncate file to 'size' length */
SPL_METHOD(SplFileObject, ftruncate)
{
  spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());
  zend_long size;

  if (zend_parse_parameters(ZEND_NUM_ARGS(), "l", &size) == FAILURE) {
    return;
  }

  if(!intern->u.file.stream) {
    zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Object not initialized");
    return;
  }

  if (!php_stream_truncate_supported(intern->u.file.stream)) {
    zend_throw_exception_ex(spl_ce_LogicException, 0, "Can't truncate file %s", intern->file_name);
    RETURN_FALSE;
  }

  RETURN_BOOL(0 == php_stream_truncate_set_size(intern->u.file.stream, size));
} /* }}} */

/* {{{ proto void SplFileObject::seek(int line_pos)
   Seek to specified line */
SPL_METHOD(SplFileObject, seek)
{
  spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());
  zend_long line_pos;

  if (zend_parse_parameters(ZEND_NUM_ARGS(), "l", &line_pos) == FAILURE) {
    return;
  }
  if(!intern->u.file.stream) {
    zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Object not initialized");
    return;
  }

  if (line_pos < 0) {
    zend_throw_exception_ex(spl_ce_LogicException, 0, "Can't seek file %s to negative line " ZEND_LONG_FMT, intern->file_name, line_pos);
    RETURN_FALSE;
  }

  spl_filesystem_file_rewind(getThis(), intern);

  while(intern->u.file.current_line_num < line_pos) {
    if (spl_filesystem_file_read_line(getThis(), intern, 1) == FAILURE) {
      break;
    }
  }
} /* }}} */

/* {{{ Function/Class/Method definitions */
ZEND_BEGIN_ARG_INFO_EX(arginfo_file_object___construct, 0, 0, 1)
  ZEND_ARG_INFO(0, file_name)
  ZEND_ARG_INFO(0, open_mode)
  ZEND_ARG_INFO(0, use_include_path)
  ZEND_ARG_INFO(0, context)
ZEND_END_ARG_INFO()

ZEND_BEGIN_ARG_INFO(arginfo_file_object_setFlags, 0)
  ZEND_ARG_INFO(0, flags)
ZEND_END_ARG_INFO()

ZEND_BEGIN_ARG_INFO(arginfo_file_object_setMaxLineLen, 0)
  ZEND_ARG_INFO(0, max_len)
ZEND_END_ARG_INFO()

ZEND_BEGIN_ARG_INFO_EX(arginfo_file_object_fgetcsv, 0, 0, 0)
  ZEND_ARG_INFO(0, delimiter)
  ZEND_ARG_INFO(0, enclosure)
  ZEND_ARG_INFO(0, escape)
ZEND_END_ARG_INFO()

ZEND_BEGIN_ARG_INFO_EX(arginfo_file_object_fputcsv, 0, 0, 1)
  ZEND_ARG_INFO(0, fields)
  ZEND_ARG_INFO(0, delimiter)
  ZEND_ARG_INFO(0, enclosure)
  ZEND_ARG_INFO(0, escape)
ZEND_END_ARG_INFO()

ZEND_BEGIN_ARG_INFO_EX(arginfo_file_object_flock, 0, 0, 1)
  ZEND_ARG_INFO(0, operation)
  ZEND_ARG_INFO(1, wouldblock)
ZEND_END_ARG_INFO()

ZEND_BEGIN_ARG_INFO_EX(arginfo_file_object_fseek, 0, 0, 1)
  ZEND_ARG_INFO(0, pos)
  ZEND_ARG_INFO(0, whence)
ZEND_END_ARG_INFO()

ZEND_BEGIN_ARG_INFO_EX(arginfo_file_object_fgetss, 0, 0, 0)
  ZEND_ARG_INFO(0, allowable_tags)
ZEND_END_ARG_INFO()

ZEND_BEGIN_ARG_INFO_EX(arginfo_file_object_fscanf, 0, 0, 1)
  ZEND_ARG_INFO(0, format)
  ZEND_ARG_VARIADIC_INFO(1, vars)
ZEND_END_ARG_INFO()

ZEND_BEGIN_ARG_INFO_EX(arginfo_file_object_fwrite, 0, 0, 1)
  ZEND_ARG_INFO(0, str)
  ZEND_ARG_INFO(0, length)
ZEND_END_ARG_INFO()

ZEND_BEGIN_ARG_INFO_EX(arginfo_file_object_fread, 0, 0, 1)
  ZEND_ARG_INFO(0, length)
ZEND_END_ARG_INFO()

ZEND_BEGIN_ARG_INFO_EX(arginfo_file_object_ftruncate, 0, 0, 1)
  ZEND_ARG_INFO(0, size)
ZEND_END_ARG_INFO()

ZEND_BEGIN_ARG_INFO_EX(arginfo_file_object_seek, 0, 0, 1)
  ZEND_ARG_INFO(0, line_pos)
ZEND_END_ARG_INFO()

static const zend_function_entry spl_SplFileObject_functions[] = {
  SPL_ME(SplFileObject, __construct,    arginfo_file_object___construct,   ZEND_ACC_PUBLIC)
  SPL_ME(SplFileObject, rewind,         arginfo_splfileinfo_void,          ZEND_ACC_PUBLIC)
  SPL_ME(SplFileObject, eof,            arginfo_splfileinfo_void,          ZEND_ACC_PUBLIC)
  SPL_ME(SplFileObject, valid,          arginfo_splfileinfo_void,          ZEND_ACC_PUBLIC)
  SPL_ME(SplFileObject, fgets,          arginfo_splfileinfo_void,          ZEND_ACC_PUBLIC)
  SPL_ME(SplFileObject, fgetcsv,        arginfo_file_object_fgetcsv,       ZEND_ACC_PUBLIC)
  SPL_ME(SplFileObject, fputcsv,        arginfo_file_object_fputcsv,       ZEND_ACC_PUBLIC)
  SPL_ME(SplFileObject, setCsvControl,  arginfo_file_object_fgetcsv,       ZEND_ACC_PUBLIC)
  SPL_ME(SplFileObject, getCsvControl,  arginfo_splfileinfo_void,          ZEND_ACC_PUBLIC)
  SPL_ME(SplFileObject, flock,          arginfo_file_object_flock,         ZEND_ACC_PUBLIC)
  SPL_ME(SplFileObject, fflush,         arginfo_splfileinfo_void,          ZEND_ACC_PUBLIC)
  SPL_ME(SplFileObject, ftell,          arginfo_splfileinfo_void,          ZEND_ACC_PUBLIC)
  SPL_ME(SplFileObject, fseek,          arginfo_file_object_fseek,         ZEND_ACC_PUBLIC)
  SPL_ME(SplFileObject, fgetc,          arginfo_splfileinfo_void,          ZEND_ACC_PUBLIC)
  SPL_ME(SplFileObject, fpassthru,      arginfo_splfileinfo_void,          ZEND_ACC_PUBLIC)
  SPL_ME(SplFileObject, fgetss,         arginfo_file_object_fgetss,        ZEND_ACC_PUBLIC)
  SPL_ME(SplFileObject, fscanf,         arginfo_file_object_fscanf,        ZEND_ACC_PUBLIC)
  SPL_ME(SplFileObject, fwrite,         arginfo_file_object_fwrite,        ZEND_ACC_PUBLIC)
  SPL_ME(SplFileObject, fread,          arginfo_file_object_fread,         ZEND_ACC_PUBLIC)
  SPL_ME(SplFileObject, fstat,          arginfo_splfileinfo_void,          ZEND_ACC_PUBLIC)
  SPL_ME(SplFileObject, ftruncate,      arginfo_file_object_ftruncate,     ZEND_ACC_PUBLIC)
  SPL_ME(SplFileObject, current,        arginfo_splfileinfo_void,          ZEND_ACC_PUBLIC)
  SPL_ME(SplFileObject, key,            arginfo_splfileinfo_void,          ZEND_ACC_PUBLIC)
  SPL_ME(SplFileObject, next,           arginfo_splfileinfo_void,          ZEND_ACC_PUBLIC)
  SPL_ME(SplFileObject, setFlags,       arginfo_file_object_setFlags,      ZEND_ACC_PUBLIC)
  SPL_ME(SplFileObject, getFlags,       arginfo_splfileinfo_void,          ZEND_ACC_PUBLIC)
  SPL_ME(SplFileObject, setMaxLineLen,  arginfo_file_object_setMaxLineLen, ZEND_ACC_PUBLIC)
  SPL_ME(SplFileObject, getMaxLineLen,  arginfo_splfileinfo_void,          ZEND_ACC_PUBLIC)
  SPL_ME(SplFileObject, hasChildren,    arginfo_splfileinfo_void,          ZEND_ACC_PUBLIC)
  SPL_ME(SplFileObject, getChildren,    arginfo_splfileinfo_void,          ZEND_ACC_PUBLIC)
  SPL_ME(SplFileObject, seek,           arginfo_file_object_seek,          ZEND_ACC_PUBLIC)
  /* mappings */
  SPL_MA(SplFileObject, getCurrentLine, SplFileObject, fgets,      arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
  SPL_MA(SplFileObject, __toString,     SplFileObject, current,    arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
  PHP_FE_END
};

ZEND_BEGIN_ARG_INFO_EX(arginfo_temp_file_object___construct, 0, 0, 0)
  ZEND_ARG_INFO(0, max_memory)
ZEND_END_ARG_INFO()

static const zend_function_entry spl_SplTempFileObject_functions[] = {
  SPL_ME(SplTempFileObject, __construct, arginfo_temp_file_object___construct,  ZEND_ACC_PUBLIC)
  PHP_FE_END
};
/* }}} */

/* {{{ PHP_MINIT_FUNCTION(spl_directory)
 */
PHP_MINIT_FUNCTION(spl_directory)
{
  REGISTER_SPL_STD_CLASS_EX(SplFileInfo, spl_filesystem_object_new, spl_SplFileInfo_functions);
  memcpy(&spl_filesystem_object_handlers, zend_get_std_object_handlers(), sizeof(zend_object_handlers));
  spl_filesystem_object_handlers.offset = XtOffsetOf(spl_filesystem_object, std);
  spl_filesystem_object_handlers.clone_obj = spl_filesystem_object_clone;
  spl_filesystem_object_handlers.cast_object = spl_filesystem_object_cast;
  spl_filesystem_object_handlers.get_debug_info  = spl_filesystem_object_get_debug_info;
  spl_filesystem_object_handlers.dtor_obj = spl_filesystem_object_destroy_object;
  spl_filesystem_object_handlers.free_obj = spl_filesystem_object_free_storage;
  spl_ce_SplFileInfo->serialize = zend_class_serialize_deny;
  spl_ce_SplFileInfo->unserialize = zend_class_unserialize_deny;


  REGISTER_SPL_SUB_CLASS_EX(DirectoryIterator, SplFileInfo, spl_filesystem_object_new, spl_DirectoryIterator_functions);
  zend_class_implements(spl_ce_DirectoryIterator, 1, zend_ce_iterator);
  REGISTER_SPL_IMPLEMENTS(DirectoryIterator, SeekableIterator);

  spl_ce_DirectoryIterator->get_iterator = spl_filesystem_dir_get_iterator;

  REGISTER_SPL_SUB_CLASS_EX(FilesystemIterator, DirectoryIterator, spl_filesystem_object_new, spl_FilesystemIterator_functions);

  REGISTER_SPL_CLASS_CONST_LONG(FilesystemIterator, "CURRENT_MODE_MASK",   SPL_FILE_DIR_CURRENT_MODE_MASK);
  REGISTER_SPL_CLASS_CONST_LONG(FilesystemIterator, "CURRENT_AS_PATHNAME", SPL_FILE_DIR_CURRENT_AS_PATHNAME);
  REGISTER_SPL_CLASS_CONST_LONG(FilesystemIterator, "CURRENT_AS_FILEINFO", SPL_FILE_DIR_CURRENT_AS_FILEINFO);
  REGISTER_SPL_CLASS_CONST_LONG(FilesystemIterator, "CURRENT_AS_SELF",     SPL_FILE_DIR_CURRENT_AS_SELF);
  REGISTER_SPL_CLASS_CONST_LONG(FilesystemIterator, "KEY_MODE_MASK",       SPL_FILE_DIR_KEY_MODE_MASK);
  REGISTER_SPL_CLASS_CONST_LONG(FilesystemIterator, "KEY_AS_PATHNAME",     SPL_FILE_DIR_KEY_AS_PATHNAME);
  REGISTER_SPL_CLASS_CONST_LONG(FilesystemIterator, "FOLLOW_SYMLINKS",     SPL_FILE_DIR_FOLLOW_SYMLINKS);
  REGISTER_SPL_CLASS_CONST_LONG(FilesystemIterator, "KEY_AS_FILENAME",     SPL_FILE_DIR_KEY_AS_FILENAME);
  REGISTER_SPL_CLASS_CONST_LONG(FilesystemIterator, "NEW_CURRENT_AND_KEY", SPL_FILE_DIR_KEY_AS_FILENAME|SPL_FILE_DIR_CURRENT_AS_FILEINFO);
  REGISTER_SPL_CLASS_CONST_LONG(FilesystemIterator, "OTHER_MODE_MASK",     SPL_FILE_DIR_OTHERS_MASK);
  REGISTER_SPL_CLASS_CONST_LONG(FilesystemIterator, "SKIP_DOTS",           SPL_FILE_DIR_SKIPDOTS);
  REGISTER_SPL_CLASS_CONST_LONG(FilesystemIterator, "UNIX_PATHS",          SPL_FILE_DIR_UNIXPATHS);

  spl_ce_FilesystemIterator->get_iterator = spl_filesystem_tree_get_iterator;

  REGISTER_SPL_SUB_CLASS_EX(RecursiveDirectoryIterator, FilesystemIterator, spl_filesystem_object_new, spl_RecursiveDirectoryIterator_functions);
  REGISTER_SPL_IMPLEMENTS(RecursiveDirectoryIterator, RecursiveIterator);

  memcpy(&spl_filesystem_object_check_handlers, &spl_filesystem_object_handlers, sizeof(zend_object_handlers));
  spl_filesystem_object_check_handlers.get_method = spl_filesystem_object_get_method_check;

#ifdef HAVE_GLOB
  REGISTER_SPL_SUB_CLASS_EX(GlobIterator, FilesystemIterator, spl_filesystem_object_new_check, spl_GlobIterator_functions);
  REGISTER_SPL_IMPLEMENTS(GlobIterator, Countable);
#endif

  REGISTER_SPL_SUB_CLASS_EX(SplFileObject, SplFileInfo, spl_filesystem_object_new_check, spl_SplFileObject_functions);
  REGISTER_SPL_IMPLEMENTS(SplFileObject, RecursiveIterator);
  REGISTER_SPL_IMPLEMENTS(SplFileObject, SeekableIterator);

  REGISTER_SPL_CLASS_CONST_LONG(SplFileObject, "DROP_NEW_LINE", SPL_FILE_OBJECT_DROP_NEW_LINE);
  REGISTER_SPL_CLASS_CONST_LONG(SplFileObject, "READ_AHEAD",    SPL_FILE_OBJECT_READ_AHEAD);
  REGISTER_SPL_CLASS_CONST_LONG(SplFileObject, "SKIP_EMPTY",    SPL_FILE_OBJECT_SKIP_EMPTY);
  REGISTER_SPL_CLASS_CONST_LONG(SplFileObject, "READ_CSV",      SPL_FILE_OBJECT_READ_CSV);

  REGISTER_SPL_SUB_CLASS_EX(SplTempFileObject, SplFileObject, spl_filesystem_object_new_check, spl_SplTempFileObject_functions);
  return SUCCESS;
}
/* }}} */

/*
 * Local variables:
 * tab-width: 4
 * c-basic-offset: 4
 * End:
 * vim600: noet sw=4 ts=4 fdm=marker
 * vim<600: noet sw=4 ts=4
 */
```

```c
/*
   +----------------------------------------------------------------------+
   | PHP Version 7                                                        |
   +----------------------------------------------------------------------+
   | Copyright (c) 1997-2018 The PHP Group                                |
   +----------------------------------------------------------------------+
   | This source file is subject to version 3.01 of the PHP license,      |
   | that is bundled with this package in the file LICENSE, and is        |
   | available through the world-wide-web at the following url:           |
   | http://www.php.net/license/3_01.txt                                  |
   | If you did not receive a copy of the PHP license and are unable to   |
   | obtain it through the world-wide-web, please send a note to          |
   | license@php.net so we can mail you a copy immediately.               |
   +----------------------------------------------------------------------+
   | Authors: Marcus Boerger <helly@php.net>                              |
   +----------------------------------------------------------------------+
 */

/* $Id$ */

#ifndef SPL_ITERATORS_H
#define SPL_ITERATORS_H

#include "php.h"
#include "php_spl.h"
#if HAVE_PCRE || HAVE_BUNDLED_PCRE
#include "ext/pcre/php_pcre.h"
#endif

#define spl_ce_Traversable   zend_ce_traversable
#define spl_ce_Iterator      zend_ce_iterator
#define spl_ce_Aggregate     zend_ce_aggregate
#define spl_ce_ArrayAccess   zend_ce_arrayaccess
#define spl_ce_Serializable  zend_ce_serializable
#define spl_ce_Countable     zend_ce_countable

extern PHPAPI zend_class_entry *spl_ce_RecursiveIterator;
extern PHPAPI zend_class_entry *spl_ce_RecursiveIteratorIterator;
extern PHPAPI zend_class_entry *spl_ce_RecursiveTreeIterator;
extern PHPAPI zend_class_entry *spl_ce_FilterIterator;
extern PHPAPI zend_class_entry *spl_ce_RecursiveFilterIterator;
extern PHPAPI zend_class_entry *spl_ce_ParentIterator;
extern PHPAPI zend_class_entry *spl_ce_SeekableIterator;
extern PHPAPI zend_class_entry *spl_ce_LimitIterator;
extern PHPAPI zend_class_entry *spl_ce_CachingIterator;
extern PHPAPI zend_class_entry *spl_ce_RecursiveCachingIterator;
extern PHPAPI zend_class_entry *spl_ce_OuterIterator;
extern PHPAPI zend_class_entry *spl_ce_IteratorIterator;
extern PHPAPI zend_class_entry *spl_ce_NoRewindIterator;
extern PHPAPI zend_class_entry *spl_ce_InfiniteIterator;
extern PHPAPI zend_class_entry *spl_ce_EmptyIterator;
extern PHPAPI zend_class_entry *spl_ce_AppendIterator;
extern PHPAPI zend_class_entry *spl_ce_RegexIterator;
extern PHPAPI zend_class_entry *spl_ce_RecursiveRegexIterator;
extern PHPAPI zend_class_entry *spl_ce_CallbackFilterIterator;
extern PHPAPI zend_class_entry *spl_ce_RecursiveCallbackFilterIterator;

PHP_MINIT_FUNCTION(spl_iterators);

PHP_FUNCTION(iterator_to_array);
PHP_FUNCTION(iterator_count);
PHP_FUNCTION(iterator_apply);

typedef enum {
  DIT_Default = 0,
  DIT_FilterIterator = DIT_Default,
  DIT_RecursiveFilterIterator = DIT_Default,
  DIT_ParentIterator = DIT_Default,
  DIT_LimitIterator,
  DIT_CachingIterator,
  DIT_RecursiveCachingIterator,
  DIT_IteratorIterator,
  DIT_NoRewindIterator,
  DIT_InfiniteIterator,
  DIT_AppendIterator,
#if HAVE_PCRE || HAVE_BUNDLED_PCRE
  DIT_RegexIterator,
  DIT_RecursiveRegexIterator,
#endif
  DIT_CallbackFilterIterator,
  DIT_RecursiveCallbackFilterIterator,
  DIT_Unknown = ~0
} dual_it_type;

typedef enum {
  RIT_Default = 0,
  RIT_RecursiveIteratorIterator = RIT_Default,
  RIT_RecursiveTreeIterator,
  RIT_Unknow = ~0
} recursive_it_it_type;

enum {
  /* public */
  CIT_CALL_TOSTRING        = 0x00000001,
  CIT_TOSTRING_USE_KEY     = 0x00000002,
  CIT_TOSTRING_USE_CURRENT = 0x00000004,
  CIT_TOSTRING_USE_INNER   = 0x00000008,
  CIT_CATCH_GET_CHILD      = 0x00000010,
  CIT_FULL_CACHE           = 0x00000100,
  CIT_PUBLIC               = 0x0000FFFF,
  /* private */
  CIT_VALID                = 0x00010000,
  CIT_HAS_CHILDREN         = 0x00020000
};

enum {
  /* public */
  REGIT_USE_KEY            = 0x00000001,
  REGIT_INVERTED           = 0x00000002
};

typedef enum {
  REGIT_MODE_MATCH,
  REGIT_MODE_GET_MATCH,
  REGIT_MODE_ALL_MATCHES,
  REGIT_MODE_SPLIT,
  REGIT_MODE_REPLACE,
  REGIT_MODE_MAX
} regex_mode;

typedef struct _spl_cbfilter_it_intern {
  zend_fcall_info       fci;
  zend_fcall_info_cache fcc;
  zend_object           *object;
} _spl_cbfilter_it_intern;

typedef struct _spl_dual_it_object {
  struct {
    zval                 zobject;
    zend_class_entry     *ce;
    zend_object          *object;
    zend_object_iterator *iterator;
  } inner;
  struct {
    zval                 data;
    zval                 key;
    zend_long            pos;
  } current;
  dual_it_type             dit_type;
  union {
    struct {
      zend_long             offset;
      zend_long             count;
    } limit;
    struct {
      zend_long             flags; /* CIT_* */
      zval             zstr;
      zval             zchildren;
      zval             zcache;
    } caching;
    struct {
      zval                  zarrayit;
      zend_object_iterator *iterator;
    } append;
#if HAVE_PCRE || HAVE_BUNDLED_PCRE
    struct {
      zend_long        flags;
      zend_long        preg_flags;
      pcre_cache_entry *pce;
      zend_string      *regex;
      regex_mode       mode;
      int              use_flags;
    } regex;
#endif
    _spl_cbfilter_it_intern *cbfilter;
  } u;
  zend_object              std;
} spl_dual_it_object;

static inline spl_dual_it_object *spl_dual_it_from_obj(zend_object *obj) /* {{{ */ {
  return (spl_dual_it_object*)((char*)(obj) - XtOffsetOf(spl_dual_it_object, std));
} /* }}} */

#define Z_SPLDUAL_IT_P(zv)  spl_dual_it_from_obj(Z_OBJ_P((zv)))

typedef int (*spl_iterator_apply_func_t)(zend_object_iterator *iter, void *puser);

PHPAPI int spl_iterator_apply(zval *obj, spl_iterator_apply_func_t apply_func, void *puser);

#endif /* SPL_ITERATORS_H */

/*
 * Local Variables:
 * c-basic-offset: 4
 * tab-width: 4
 * End:
 * vim600: fdm=marker
 * vim: noet sw=4 ts=4
 */
```

```c
/*
   +----------------------------------------------------------------------+
   | PHP Version 7                                                        |
   +----------------------------------------------------------------------+
   | Copyright (c) 1997-2018 The PHP Group                                |
   +----------------------------------------------------------------------+
   | This source file is subject to version 3.01 of the PHP license,      |
   | that is bundled with this package in the file LICENSE, and is        |
   | available through the world-wide-web at the following url:           |
   | http://www.php.net/license/3_01.txt                                  |
   | If you did not receive a copy of the PHP license and are unable to   |
   | obtain it through the world-wide-web, please send a note to          |
   | license@php.net so we can mail you a copy immediately.               |
   +----------------------------------------------------------------------+
   | Authors: Marcus Boerger <helly@php.net>                              |
   +----------------------------------------------------------------------+
 */

/* $Id$ */

#ifdef HAVE_CONFIG_H
# include "config.h"
#endif

#include "php.h"
#include "php_ini.h"
#include "ext/standard/info.h"
#include "ext/standard/php_var.h"
#include "zend_smart_str.h"
#include "zend_interfaces.h"
#include "zend_exceptions.h"

#include "php_spl.h"
#include "spl_functions.h"
#include "spl_engine.h"
#include "spl_iterators.h"
#include "spl_array.h"
#include "spl_exceptions.h"

zend_object_handlers spl_handler_ArrayObject;
PHPAPI zend_class_entry  *spl_ce_ArrayObject;

zend_object_handlers spl_handler_ArrayIterator;
PHPAPI zend_class_entry  *spl_ce_ArrayIterator;
PHPAPI zend_class_entry  *spl_ce_RecursiveArrayIterator;

#define SPL_ARRAY_STD_PROP_LIST      0x00000001
#define SPL_ARRAY_ARRAY_AS_PROPS     0x00000002
#define SPL_ARRAY_CHILD_ARRAYS_ONLY  0x00000004
#define SPL_ARRAY_OVERLOADED_REWIND  0x00010000
#define SPL_ARRAY_OVERLOADED_VALID   0x00020000
#define SPL_ARRAY_OVERLOADED_KEY     0x00040000
#define SPL_ARRAY_OVERLOADED_CURRENT 0x00080000
#define SPL_ARRAY_OVERLOADED_NEXT    0x00100000
#define SPL_ARRAY_IS_SELF            0x01000000
#define SPL_ARRAY_USE_OTHER          0x02000000
#define SPL_ARRAY_INT_MASK           0xFFFF0000
#define SPL_ARRAY_CLONE_MASK         0x0100FFFF

#define SPL_ARRAY_METHOD_NO_ARG       0
#define SPL_ARRAY_METHOD_USE_ARG        1
#define SPL_ARRAY_METHOD_MAY_USER_ARG     2

typedef struct _spl_array_object {
  zval              array;
  uint32_t          ht_iter;
  int               ar_flags;
  unsigned char   nApplyCount;
  zend_function     *fptr_offset_get;
  zend_function     *fptr_offset_set;
  zend_function     *fptr_offset_has;
  zend_function     *fptr_offset_del;
  zend_function     *fptr_count;
  zend_class_entry* ce_get_iterator;
  zend_object       std;
} spl_array_object;

static inline spl_array_object *spl_array_from_obj(zend_object *obj) /* {{{ */ {
  return (spl_array_object*)((char*)(obj) - XtOffsetOf(spl_array_object, std));
}
/* }}} */

#define Z_SPLARRAY_P(zv)  spl_array_from_obj(Z_OBJ_P((zv)))

static inline HashTable **spl_array_get_hash_table_ptr(spl_array_object* intern) { /* {{{ */
  //??? TODO: Delay duplication for arrays; only duplicate for write operations
  if (intern->ar_flags & SPL_ARRAY_IS_SELF) {
    if (!intern->std.properties) {
      rebuild_object_properties(&intern->std);
    }
    return &intern->std.properties;
  } else if (intern->ar_flags & SPL_ARRAY_USE_OTHER) {
    spl_array_object *other = Z_SPLARRAY_P(&intern->array);
    return spl_array_get_hash_table_ptr(other);
  } else if (Z_TYPE(intern->array) == IS_ARRAY) {
    return &Z_ARRVAL(intern->array);
  } else {
    zend_object *obj = Z_OBJ(intern->array);
    if (!obj->properties) {
      rebuild_object_properties(obj);
    } else if (GC_REFCOUNT(obj->properties) > 1) {
      if (EXPECTED(!(GC_FLAGS(obj->properties) & IS_ARRAY_IMMUTABLE))) {
        GC_DELREF(obj->properties);
      }
      obj->properties = zend_array_dup(obj->properties);
    }
    return &obj->properties;
  }
}
/* }}} */

static inline HashTable *spl_array_get_hash_table(spl_array_object* intern) { /* {{{ */
  return *spl_array_get_hash_table_ptr(intern);
}
/* }}} */

static inline void spl_array_replace_hash_table(spl_array_object* intern, HashTable *ht) { /* {{{ */
  HashTable **ht_ptr = spl_array_get_hash_table_ptr(intern);
  zend_array_destroy(*ht_ptr);
  *ht_ptr = ht;
}
/* }}} */

static inline zend_bool spl_array_is_object(spl_array_object *intern) /* {{{ */
{
  while (intern->ar_flags & SPL_ARRAY_USE_OTHER) {
    intern = Z_SPLARRAY_P(&intern->array);
  }
  return (intern->ar_flags & SPL_ARRAY_IS_SELF) || Z_TYPE(intern->array) == IS_OBJECT;
}
/* }}} */

static int spl_array_skip_protected(spl_array_object *intern, HashTable *aht);

static zend_never_inline void spl_array_create_ht_iter(HashTable *ht, spl_array_object* intern) /* {{{ */
{
  intern->ht_iter = zend_hash_iterator_add(ht, ht->nInternalPointer);
  zend_hash_internal_pointer_reset_ex(ht, &EG(ht_iterators)[intern->ht_iter].pos);
  spl_array_skip_protected(intern, ht);
}
/* }}} */

static zend_always_inline uint32_t *spl_array_get_pos_ptr(HashTable *ht, spl_array_object* intern) /* {{{ */
{
  if (UNEXPECTED(intern->ht_iter == (uint32_t)-1)) {
    spl_array_create_ht_iter(ht, intern);
  }
  return &EG(ht_iterators)[intern->ht_iter].pos;
}
/* }}} */

/* {{{ spl_array_object_free_storage */
static void spl_array_object_free_storage(zend_object *object)
{
  spl_array_object *intern = spl_array_from_obj(object);

  if (intern->ht_iter != (uint32_t) -1) {
    zend_hash_iterator_del(intern->ht_iter);
  }

  zend_object_std_dtor(&intern->std);

  zval_ptr_dtor(&intern->array);
}
/* }}} */

zend_object_iterator *spl_array_get_iterator(zend_class_entry *ce, zval *object, int by_ref);

/* {{{ spl_array_object_new_ex */
static zend_object *spl_array_object_new_ex(zend_class_entry *class_type, zval *orig, int clone_orig)
{
  spl_array_object *intern;
  zend_class_entry *parent = class_type;
  int inherited = 0;

  intern = zend_object_alloc(sizeof(spl_array_object), parent);

  zend_object_std_init(&intern->std, class_type);
  object_properties_init(&intern->std, class_type);

  intern->ar_flags = 0;
  intern->ce_get_iterator = spl_ce_ArrayIterator;
  if (orig) {
    spl_array_object *other = Z_SPLARRAY_P(orig);

    intern->ar_flags &= ~ SPL_ARRAY_CLONE_MASK;
    intern->ar_flags |= (other->ar_flags & SPL_ARRAY_CLONE_MASK);
    intern->ce_get_iterator = other->ce_get_iterator;
    if (clone_orig) {
      if (other->ar_flags & SPL_ARRAY_IS_SELF) {
        ZVAL_UNDEF(&intern->array);
      } else if (Z_OBJ_HT_P(orig) == &spl_handler_ArrayObject) {
        ZVAL_ARR(&intern->array,
          zend_array_dup(spl_array_get_hash_table(other)));
      } else {
        ZEND_ASSERT(Z_OBJ_HT_P(orig) == &spl_handler_ArrayIterator);
        ZVAL_COPY(&intern->array, orig);
        intern->ar_flags |= SPL_ARRAY_USE_OTHER;
      }
    } else {
      ZVAL_COPY(&intern->array, orig);
      intern->ar_flags |= SPL_ARRAY_USE_OTHER;
    }
  } else {
    array_init(&intern->array);
  }

  while (parent) {
    if (parent == spl_ce_ArrayIterator || parent == spl_ce_RecursiveArrayIterator) {
      intern->std.handlers = &spl_handler_ArrayIterator;
      class_type->get_iterator = spl_array_get_iterator;
      break;
    } else if (parent == spl_ce_ArrayObject) {
      intern->std.handlers = &spl_handler_ArrayObject;
      break;
    }
    parent = parent->parent;
    inherited = 1;
  }
  if (!parent) { /* this must never happen */
    php_error_docref(NULL, E_COMPILE_ERROR, "Internal compiler error, Class is not child of ArrayObject or ArrayIterator");
  }
  if (inherited) {
    intern->fptr_offset_get = zend_hash_str_find_ptr(&class_type->function_table, "offsetget", sizeof("offsetget") - 1);
    if (intern->fptr_offset_get->common.scope == parent) {
      intern->fptr_offset_get = NULL;
    }
    intern->fptr_offset_set = zend_hash_str_find_ptr(&class_type->function_table, "offsetset", sizeof("offsetset") - 1);
    if (intern->fptr_offset_set->common.scope == parent) {
      intern->fptr_offset_set = NULL;
    }
    intern->fptr_offset_has = zend_hash_str_find_ptr(&class_type->function_table, "offsetexists", sizeof("offsetexists") - 1);
    if (intern->fptr_offset_has->common.scope == parent) {
      intern->fptr_offset_has = NULL;
    }
    intern->fptr_offset_del = zend_hash_str_find_ptr(&class_type->function_table, "offsetunset",  sizeof("offsetunset") - 1);
    if (intern->fptr_offset_del->common.scope == parent) {
      intern->fptr_offset_del = NULL;
    }
    intern->fptr_count = zend_hash_str_find_ptr(&class_type->function_table, "count", sizeof("count") - 1);
    if (intern->fptr_count->common.scope == parent) {
      intern->fptr_count = NULL;
    }
  }
  /* Cache iterator functions if ArrayIterator or derived. Check current's */
  /* cache since only current is always required */
  if (intern->std.handlers == &spl_handler_ArrayIterator) {
    if (!class_type->iterator_funcs.zf_current) {
      class_type->iterator_funcs.zf_rewind = zend_hash_str_find_ptr(&class_type->function_table, "rewind", sizeof("rewind") - 1);
      class_type->iterator_funcs.zf_valid = zend_hash_str_find_ptr(&class_type->function_table, "valid", sizeof("valid") - 1);
      class_type->iterator_funcs.zf_key = zend_hash_str_find_ptr(&class_type->function_table, "key", sizeof("key") - 1);
      class_type->iterator_funcs.zf_current = zend_hash_str_find_ptr(&class_type->function_table, "current", sizeof("current") - 1);
      class_type->iterator_funcs.zf_next = zend_hash_str_find_ptr(&class_type->function_table, "next", sizeof("next") - 1);
    }
    if (inherited) {
      if (class_type->iterator_funcs.zf_rewind->common.scope  != parent) intern->ar_flags |= SPL_ARRAY_OVERLOADED_REWIND;
      if (class_type->iterator_funcs.zf_valid->common.scope   != parent) intern->ar_flags |= SPL_ARRAY_OVERLOADED_VALID;
      if (class_type->iterator_funcs.zf_key->common.scope     != parent) intern->ar_flags |= SPL_ARRAY_OVERLOADED_KEY;
      if (class_type->iterator_funcs.zf_current->common.scope != parent) intern->ar_flags |= SPL_ARRAY_OVERLOADED_CURRENT;
      if (class_type->iterator_funcs.zf_next->common.scope    != parent) intern->ar_flags |= SPL_ARRAY_OVERLOADED_NEXT;
    }
  }

  intern->ht_iter = (uint32_t)-1;
  return &intern->std;
}
/* }}} */

/* {{{ spl_array_object_new */
static zend_object *spl_array_object_new(zend_class_entry *class_type)
{
  return spl_array_object_new_ex(class_type, NULL, 0);
}
/* }}} */

/* {{{ spl_array_object_clone */
static zend_object *spl_array_object_clone(zval *zobject)
{
  zend_object *old_object;
  zend_object *new_object;

  old_object = Z_OBJ_P(zobject);
  new_object = spl_array_object_new_ex(old_object->ce, zobject, 1);

  zend_objects_clone_members(new_object, old_object);

  return new_object;
}
/* }}} */

static zval *spl_array_get_dimension_ptr(int check_inherited, spl_array_object *intern, zval *offset, int type) /* {{{ */
{
  zval *retval;
  zend_long index;
  zend_string *offset_key;
  HashTable *ht = spl_array_get_hash_table(intern);

  if (!offset || Z_ISUNDEF_P(offset) || !ht) {
    return &EG(uninitialized_zval);
  }

  if ((type == BP_VAR_W || type == BP_VAR_RW) && intern->nApplyCount > 0) {
    zend_error(E_WARNING, "Modification of ArrayObject during sorting is prohibited");
    return &EG(error_zval);
  }

try_again:
  switch (Z_TYPE_P(offset)) {
  case IS_NULL:
     offset_key = ZSTR_EMPTY_ALLOC();
     goto fetch_dim_string;
  case IS_STRING:
     offset_key = Z_STR_P(offset);
fetch_dim_string:
    retval = zend_symtable_find(ht, offset_key);
    if (retval) {
      if (Z_TYPE_P(retval) == IS_INDIRECT) {
        retval = Z_INDIRECT_P(retval);
        if (Z_TYPE_P(retval) == IS_UNDEF) {
          switch (type) {
            case BP_VAR_R:
              zend_error(E_NOTICE, "Undefined index: %s", ZSTR_VAL(offset_key));
            case BP_VAR_UNSET:
            case BP_VAR_IS:
              retval = &EG(uninitialized_zval);
              break;
            case BP_VAR_RW:
              zend_error(E_NOTICE,"Undefined index: %s", ZSTR_VAL(offset_key));
            case BP_VAR_W: {
              ZVAL_NULL(retval);
            }
          }
        }
      }
    } else {
      switch (type) {
        case BP_VAR_R:
          zend_error(E_NOTICE, "Undefined index: %s", ZSTR_VAL(offset_key));
        case BP_VAR_UNSET:
        case BP_VAR_IS:
          retval = &EG(uninitialized_zval);
          break;
        case BP_VAR_RW:
          zend_error(E_NOTICE,"Undefined index: %s", ZSTR_VAL(offset_key));
        case BP_VAR_W: {
            zval value;
          ZVAL_NULL(&value);
            retval = zend_symtable_update(ht, offset_key, &value);
        }
      }
    }
    return retval;
  case IS_RESOURCE:
    zend_error(E_NOTICE, "Resource ID#%d used as offset, casting to integer (%d)", Z_RES_P(offset)->handle, Z_RES_P(offset)->handle);
    index = Z_RES_P(offset)->handle;
    goto num_index;
  case IS_DOUBLE:
    index = (zend_long)Z_DVAL_P(offset);
    goto num_index;
  case IS_FALSE:
    index = 0;
    goto num_index;
  case IS_TRUE:
    index = 1;
    goto num_index;
  case IS_LONG:
    index = Z_LVAL_P(offset);
num_index:
    if ((retval = zend_hash_index_find(ht, index)) == NULL) {
      switch (type) {
        case BP_VAR_R:
          zend_error(E_NOTICE, "Undefined offset: " ZEND_LONG_FMT, index);
        case BP_VAR_UNSET:
        case BP_VAR_IS:
          retval = &EG(uninitialized_zval);
          break;
        case BP_VAR_RW:
          zend_error(E_NOTICE, "Undefined offset: " ZEND_LONG_FMT, index);
        case BP_VAR_W: {
            zval value;
          ZVAL_UNDEF(&value);
          retval = zend_hash_index_update(ht, index, &value);
         }
      }
    }
    return retval;
  case IS_REFERENCE:
    ZVAL_DEREF(offset);
    goto try_again;
  default:
    zend_error(E_WARNING, "Illegal offset type");
    return (type == BP_VAR_W || type == BP_VAR_RW) ?
      &EG(error_zval) : &EG(uninitialized_zval);
  }
} /* }}} */

static int spl_array_has_dimension(zval *object, zval *offset, int check_empty);

static zval *spl_array_read_dimension_ex(int check_inherited, zval *object, zval *offset, int type, zval *rv) /* {{{ */
{
  spl_array_object *intern = Z_SPLARRAY_P(object);
  zval *ret;

  if (check_inherited &&
      (intern->fptr_offset_get || (type == BP_VAR_IS && intern->fptr_offset_has))) {
    if (type == BP_VAR_IS) {
      if (!spl_array_has_dimension(object, offset, 0)) {
        return &EG(uninitialized_zval);
      }
    }

    if (intern->fptr_offset_get) {
      zval tmp;
      if (!offset) {
        ZVAL_UNDEF(&tmp);
        offset = &tmp;
      } else {
        SEPARATE_ARG_IF_REF(offset);
      }
      zend_call_method_with_1_params(object, Z_OBJCE_P(object), &intern->fptr_offset_get, "offsetGet", rv, offset);
      zval_ptr_dtor(offset);

      if (!Z_ISUNDEF_P(rv)) {
        return rv;
      }
      return &EG(uninitialized_zval);
    }
  }

  ret = spl_array_get_dimension_ptr(check_inherited, intern, offset, type);

  /* When in a write context,
   * ZE has to be fooled into thinking this is in a reference set
   * by separating (if necessary) and returning as IS_REFERENCE (with refcount == 1)
   */

  if ((type == BP_VAR_W || type == BP_VAR_RW || type == BP_VAR_UNSET) &&
      !Z_ISREF_P(ret) &&
      EXPECTED(ret != &EG(uninitialized_zval))) {
    ZVAL_NEW_REF(ret, ret);
  }

  return ret;
} /* }}} */

static zval *spl_array_read_dimension(zval *object, zval *offset, int type, zval *rv) /* {{{ */
{
  return spl_array_read_dimension_ex(1, object, offset, type, rv);
} /* }}} */

static void spl_array_write_dimension_ex(int check_inherited, zval *object, zval *offset, zval *value) /* {{{ */
{
  spl_array_object *intern = Z_SPLARRAY_P(object);
  zend_long index;
  HashTable *ht;

  if (check_inherited && intern->fptr_offset_set) {
    zval tmp;

    if (!offset) {
      ZVAL_NULL(&tmp);
      offset = &tmp;
    } else {
      SEPARATE_ARG_IF_REF(offset);
    }
    zend_call_method_with_2_params(object, Z_OBJCE_P(object), &intern->fptr_offset_set, "offsetSet", NULL, offset, value);
    zval_ptr_dtor(offset);
    return;
  }

  if (intern->nApplyCount > 0) {
    zend_error(E_WARNING, "Modification of ArrayObject during sorting is prohibited");
    return;
  }

  Z_TRY_ADDREF_P(value);
  if (!offset) {
    ht = spl_array_get_hash_table(intern);
    zend_hash_next_index_insert(ht, value);
    return;
  }

try_again:
  switch (Z_TYPE_P(offset)) {
    case IS_STRING:
      ht = spl_array_get_hash_table(intern);
      zend_symtable_update_ind(ht, Z_STR_P(offset), value);
      return;
    case IS_DOUBLE:
      index = (zend_long)Z_DVAL_P(offset);
      goto num_index;
    case IS_RESOURCE:
      index = Z_RES_HANDLE_P(offset);
      goto num_index;
    case IS_FALSE:
      index = 0;
      goto num_index;
    case IS_TRUE:
      index = 1;
      goto num_index;
    case IS_LONG:
      index = Z_LVAL_P(offset);
num_index:
      ht = spl_array_get_hash_table(intern);
      zend_hash_index_update(ht, index, value);
      return;
    case IS_NULL:
      ht = spl_array_get_hash_table(intern);
      zend_hash_next_index_insert(ht, value);
      return;
    case IS_REFERENCE:
      ZVAL_DEREF(offset);
      goto try_again;
    default:
      zend_error(E_WARNING, "Illegal offset type");
      zval_ptr_dtor(value);
      return;
  }
} /* }}} */

static void spl_array_write_dimension(zval *object, zval *offset, zval *value) /* {{{ */
{
  spl_array_write_dimension_ex(1, object, offset, value);
} /* }}} */

static void spl_array_unset_dimension_ex(int check_inherited, zval *object, zval *offset) /* {{{ */
{
  zend_long index;
  HashTable *ht;
  spl_array_object *intern = Z_SPLARRAY_P(object);

  if (check_inherited && intern->fptr_offset_del) {
    SEPARATE_ARG_IF_REF(offset);
    zend_call_method_with_1_params(object, Z_OBJCE_P(object), &intern->fptr_offset_del, "offsetUnset", NULL, offset);
    zval_ptr_dtor(offset);
    return;
  }

  if (intern->nApplyCount > 0) {
    zend_error(E_WARNING, "Modification of ArrayObject during sorting is prohibited");
    return;
  }

try_again:
  switch (Z_TYPE_P(offset)) {
  case IS_STRING:
    ht = spl_array_get_hash_table(intern);
    if (ht == &EG(symbol_table)) {
      if (zend_delete_global_variable(Z_STR_P(offset))) {
        zend_error(E_NOTICE,"Undefined index: %s", Z_STRVAL_P(offset));
      }
    } else {
      zval *data = zend_symtable_find(ht, Z_STR_P(offset));

      if (data) {
        if (Z_TYPE_P(data) == IS_INDIRECT) {
          data = Z_INDIRECT_P(data);
          if (Z_TYPE_P(data) == IS_UNDEF) {
            zend_error(E_NOTICE,"Undefined index: %s", Z_STRVAL_P(offset));
          } else {
            zval_ptr_dtor(data);
            ZVAL_UNDEF(data);
            HT_FLAGS(ht) |= HASH_FLAG_HAS_EMPTY_IND;
            zend_hash_move_forward_ex(ht, spl_array_get_pos_ptr(ht, intern));
            if (spl_array_is_object(intern)) {
              spl_array_skip_protected(intern, ht);
            }
          }
        } else if (zend_symtable_del(ht, Z_STR_P(offset)) == FAILURE) {
          zend_error(E_NOTICE,"Undefined index: %s", Z_STRVAL_P(offset));
        }
      } else {
        zend_error(E_NOTICE,"Undefined index: %s", Z_STRVAL_P(offset));
      }
    }
    break;
  case IS_DOUBLE:
    index = (zend_long)Z_DVAL_P(offset);
    goto num_index;
  case IS_RESOURCE:
    index = Z_RES_HANDLE_P(offset);
    goto num_index;
  case IS_FALSE:
    index = 0;
    goto num_index;
  case IS_TRUE:
    index = 1;
    goto num_index;
  case IS_LONG:
    index = Z_LVAL_P(offset);
num_index:
    ht = spl_array_get_hash_table(intern);
    if (zend_hash_index_del(ht, index) == FAILURE) {
      zend_error(E_NOTICE,"Undefined offset: " ZEND_LONG_FMT, index);
    }
    break;
  case IS_REFERENCE:
    ZVAL_DEREF(offset);
    goto try_again;
  default:
    zend_error(E_WARNING, "Illegal offset type");
    return;
  }
} /* }}} */

static void spl_array_unset_dimension(zval *object, zval *offset) /* {{{ */
{
  spl_array_unset_dimension_ex(1, object, offset);
} /* }}} */

static int spl_array_has_dimension_ex(int check_inherited, zval *object, zval *offset, int check_empty) /* {{{ */
{
  spl_array_object *intern = Z_SPLARRAY_P(object);
  zend_long index;
  zval rv, *value = NULL, *tmp;

  if (check_inherited && intern->fptr_offset_has) {
    SEPARATE_ARG_IF_REF(offset);
    zend_call_method_with_1_params(object, Z_OBJCE_P(object), &intern->fptr_offset_has, "offsetExists", &rv, offset);
    zval_ptr_dtor(offset);

    if (!Z_ISUNDEF(rv) && zend_is_true(&rv)) {
      zval_ptr_dtor(&rv);
      if (check_empty != 1) {
        return 1;
      } else if (intern->fptr_offset_get) {
        value = spl_array_read_dimension_ex(1, object, offset, BP_VAR_R, &rv);
      }
    } else {
      zval_ptr_dtor(&rv);
      return 0;
    }
  }

  if (!value) {
    HashTable *ht = spl_array_get_hash_table(intern);

try_again:
    switch (Z_TYPE_P(offset)) {
      case IS_STRING:
        if ((tmp = zend_symtable_find(ht, Z_STR_P(offset))) != NULL) {
          if (check_empty == 2) {
            return 1;
          }
        } else {
          return 0;
        }
        break;

      case IS_DOUBLE:
        index = (zend_long)Z_DVAL_P(offset);
        goto num_index;
      case IS_RESOURCE:
        index = Z_RES_HANDLE_P(offset);
        goto num_index;
      case IS_FALSE:
        index = 0;
        goto num_index;
      case IS_TRUE:
        index = 1;
        goto num_index;
      case IS_LONG:
        index = Z_LVAL_P(offset);
num_index:
        if ((tmp = zend_hash_index_find(ht, index)) != NULL) {
          if (check_empty == 2) {
            return 1;
          }
        } else {
          return 0;
        }
        break;
      case IS_REFERENCE:
        ZVAL_DEREF(offset);
        goto try_again;
      default:
        zend_error(E_WARNING, "Illegal offset type");
        return 0;
    }

    if (check_empty && check_inherited && intern->fptr_offset_get) {
      value = spl_array_read_dimension_ex(1, object, offset, BP_VAR_R, &rv);
    } else {
      value = tmp;
    }
  }

  {
    zend_bool result = check_empty ? zend_is_true(value) : Z_TYPE_P(value) != IS_NULL;
    if (value == &rv) {
      zval_ptr_dtor(&rv);
    }
    return result;
  }
} /* }}} */

static int spl_array_has_dimension(zval *object, zval *offset, int check_empty) /* {{{ */
{
  return spl_array_has_dimension_ex(1, object, offset, check_empty);
} /* }}} */

/* {{{ spl_array_object_verify_pos_ex */
static inline int spl_array_object_verify_pos_ex(spl_array_object *object, HashTable *ht, const char *msg_prefix)
{
  if (!ht) {
    php_error_docref(NULL, E_NOTICE, "%sArray was modified outside object and is no longer an array", msg_prefix);
    return FAILURE;
  }

  return SUCCESS;
} /* }}} */

/* {{{ spl_array_object_verify_pos */
static inline int spl_array_object_verify_pos(spl_array_object *object, HashTable *ht)
{
  return spl_array_object_verify_pos_ex(object, ht, "");
} /* }}} */

/* {{{ proto bool ArrayObject::offsetExists(mixed $index)
       proto bool ArrayIterator::offsetExists(mixed $index)
   Returns whether the requested $index exists. */
SPL_METHOD(Array, offsetExists)
{
  zval *index;
  if (zend_parse_parameters(ZEND_NUM_ARGS(), "z", &index) == FAILURE) {
    return;
  }
  RETURN_BOOL(spl_array_has_dimension_ex(0, getThis(), index, 2));
} /* }}} */

/* {{{ proto mixed ArrayObject::offsetGet(mixed $index)
       proto mixed ArrayIterator::offsetGet(mixed $index)
   Returns the value at the specified $index. */
SPL_METHOD(Array, offsetGet)
{
  zval *value, *index;
  if (zend_parse_parameters(ZEND_NUM_ARGS(), "z", &index) == FAILURE) {
    return;
  }
  value = spl_array_read_dimension_ex(0, getThis(), index, BP_VAR_R, return_value);
  if (value != return_value) {
    ZVAL_DEREF(value);
    ZVAL_COPY(return_value, value);
  }
} /* }}} */

/* {{{ proto void ArrayObject::offsetSet(mixed $index, mixed $newval)
       proto void ArrayIterator::offsetSet(mixed $index, mixed $newval)
   Sets the value at the specified $index to $newval. */
SPL_METHOD(Array, offsetSet)
{
  zval *index, *value;
  if (zend_parse_parameters(ZEND_NUM_ARGS(), "zz", &index, &value) == FAILURE) {
    return;
  }
  spl_array_write_dimension_ex(0, getThis(), index, value);
} /* }}} */

void spl_array_iterator_append(zval *object, zval *append_value) /* {{{ */
{
  spl_array_object *intern = Z_SPLARRAY_P(object);
  HashTable *aht = spl_array_get_hash_table(intern);

  if (!aht) {
    php_error_docref(NULL, E_NOTICE, "Array was modified outside object and is no longer an array");
    return;
  }

  if (spl_array_is_object(intern)) {
    zend_throw_error(NULL, "Cannot append properties to objects, use %s::offsetSet() instead", ZSTR_VAL(Z_OBJCE_P(object)->name));
    return;
  }

  spl_array_write_dimension(object, NULL, append_value);
} /* }}} */

/* {{{ proto void ArrayObject::append(mixed $newval)
       proto void ArrayIterator::append(mixed $newval)
   Appends the value (cannot be called for objects). */
SPL_METHOD(Array, append)
{
  zval *value;

  if (zend_parse_parameters(ZEND_NUM_ARGS(), "z", &value) == FAILURE) {
    return;
  }
  spl_array_iterator_append(getThis(), value);
} /* }}} */

/* {{{ proto void ArrayObject::offsetUnset(mixed $index)
       proto void ArrayIterator::offsetUnset(mixed $index)
   Unsets the value at the specified $index. */
SPL_METHOD(Array, offsetUnset)
{
  zval *index;
  if (zend_parse_parameters(ZEND_NUM_ARGS(), "z", &index) == FAILURE) {
    return;
  }
  spl_array_unset_dimension_ex(0, getThis(), index);
} /* }}} */

/* {{{ proto array ArrayObject::getArrayCopy()
      proto array ArrayIterator::getArrayCopy()
   Return a copy of the contained array */
SPL_METHOD(Array, getArrayCopy)
{
  zval *object = getThis();
  spl_array_object *intern = Z_SPLARRAY_P(object);

  RETURN_ARR(zend_array_dup(spl_array_get_hash_table(intern)));
} /* }}} */

static HashTable *spl_array_get_properties(zval *object) /* {{{ */
{
  spl_array_object *intern = Z_SPLARRAY_P(object);

  if (intern->ar_flags & SPL_ARRAY_STD_PROP_LIST) {
    if (!intern->std.properties) {
      rebuild_object_properties(&intern->std);
    }
    return intern->std.properties;
  }

  return spl_array_get_hash_table(intern);
} /* }}} */

static HashTable* spl_array_get_debug_info(zval *obj, int *is_temp) /* {{{ */
{
  zval *storage;
  zend_string *zname;
  zend_class_entry *base;
  spl_array_object *intern = Z_SPLARRAY_P(obj);

  if (!intern->std.properties) {
    rebuild_object_properties(&intern->std);
  }

  if (intern->ar_flags & SPL_ARRAY_IS_SELF) {
    *is_temp = 0;
    return intern->std.properties;
  } else {
    HashTable *debug_info;
    *is_temp = 1;

    debug_info = zend_new_array(zend_hash_num_elements(intern->std.properties) + 1);
    zend_hash_copy(debug_info, intern->std.properties, (copy_ctor_func_t) zval_add_ref);

    storage = &intern->array;
    Z_TRY_ADDREF_P(storage);

    base = Z_OBJ_HT_P(obj) == &spl_handler_ArrayIterator
      ? spl_ce_ArrayIterator : spl_ce_ArrayObject;
    zname = spl_gen_private_prop_name(base, "storage", sizeof("storage")-1);
    zend_symtable_update(debug_info, zname, storage);
    zend_string_release(zname);

    return debug_info;
  }
}
/* }}} */

static HashTable *spl_array_get_gc(zval *obj, zval **gc_data, int *gc_data_count) /* {{{ */
{
  spl_array_object *intern = Z_SPLARRAY_P(obj);
  *gc_data = &intern->array;
  *gc_data_count = 1;
  return zend_std_get_properties(obj);
}
/* }}} */

static zval *spl_array_read_property(zval *object, zval *member, int type, void **cache_slot, zval *rv) /* {{{ */
{
  spl_array_object *intern = Z_SPLARRAY_P(object);

  if ((intern->ar_flags & SPL_ARRAY_ARRAY_AS_PROPS) != 0
    && !std_object_handlers.has_property(object, member, 2, NULL)) {
    return spl_array_read_dimension(object, member, type, rv);
  }
  return std_object_handlers.read_property(object, member, type, cache_slot, rv);
} /* }}} */

static void spl_array_write_property(zval *object, zval *member, zval *value, void **cache_slot) /* {{{ */
{
  spl_array_object *intern = Z_SPLARRAY_P(object);

  if ((intern->ar_flags & SPL_ARRAY_ARRAY_AS_PROPS) != 0
  && !std_object_handlers.has_property(object, member, 2, NULL)) {
    spl_array_write_dimension(object, member, value);
    return;
  }
  std_object_handlers.write_property(object, member, value, cache_slot);
} /* }}} */

static zval *spl_array_get_property_ptr_ptr(zval *object, zval *member, int type, void **cache_slot) /* {{{ */
{
  spl_array_object *intern = Z_SPLARRAY_P(object);

  if ((intern->ar_flags & SPL_ARRAY_ARRAY_AS_PROPS) != 0
    && !std_object_handlers.has_property(object, member, 2, NULL)) {
    /* If object has offsetGet() overridden, then fallback to read_property,
     * which will call offsetGet(). */
    if (intern->fptr_offset_get) {
      return NULL;
    }
    return spl_array_get_dimension_ptr(1, intern, member, type);
  }
  return std_object_handlers.get_property_ptr_ptr(object, member, type, cache_slot);
} /* }}} */

static int spl_array_has_property(zval *object, zval *member, int has_set_exists, void **cache_slot) /* {{{ */
{
  spl_array_object *intern = Z_SPLARRAY_P(object);

  if ((intern->ar_flags & SPL_ARRAY_ARRAY_AS_PROPS) != 0
    && !std_object_handlers.has_property(object, member, 2, NULL)) {
    return spl_array_has_dimension(object, member, has_set_exists);
  }
  return std_object_handlers.has_property(object, member, has_set_exists, cache_slot);
} /* }}} */

static void spl_array_unset_property(zval *object, zval *member, void **cache_slot) /* {{{ */
{
  spl_array_object *intern = Z_SPLARRAY_P(object);

  if ((intern->ar_flags & SPL_ARRAY_ARRAY_AS_PROPS) != 0
    && !std_object_handlers.has_property(object, member, 2, NULL)) {
    spl_array_unset_dimension(object, member);
    return;
  }
  std_object_handlers.unset_property(object, member, cache_slot);
} /* }}} */

static int spl_array_compare_objects(zval *o1, zval *o2) /* {{{ */
{
  HashTable     *ht1,
          *ht2;
  spl_array_object  *intern1,
          *intern2;
  int         result  = 0;

  intern1 = Z_SPLARRAY_P(o1);
  intern2 = Z_SPLARRAY_P(o2);
  ht1   = spl_array_get_hash_table(intern1);
  ht2   = spl_array_get_hash_table(intern2);

  result = zend_compare_symbol_tables(ht1, ht2);
  /* if we just compared std.properties, don't do it again */
  if (result == 0 &&
      !(ht1 == intern1->std.properties && ht2 == intern2->std.properties)) {
    result = std_object_handlers.compare_objects(o1, o2);
  }
  return result;
} /* }}} */

static int spl_array_skip_protected(spl_array_object *intern, HashTable *aht) /* {{{ */
{
  zend_string *string_key;
  zend_ulong num_key;
  zval *data;

  if (spl_array_is_object(intern)) {
    uint32_t *pos_ptr = spl_array_get_pos_ptr(aht, intern);

    do {
      if (zend_hash_get_current_key_ex(aht, &string_key, &num_key, pos_ptr) == HASH_KEY_IS_STRING) {
        data = zend_hash_get_current_data_ex(aht, pos_ptr);
        if (data && Z_TYPE_P(data) == IS_INDIRECT &&
            Z_TYPE_P(data = Z_INDIRECT_P(data)) == IS_UNDEF) {
          /* skip */
        } else if (!ZSTR_LEN(string_key) || ZSTR_VAL(string_key)[0]) {
          return SUCCESS;
        }
      } else {
        return SUCCESS;
      }
      if (zend_hash_has_more_elements_ex(aht, pos_ptr) != SUCCESS) {
        return FAILURE;
      }
      zend_hash_move_forward_ex(aht, pos_ptr);
    } while (1);
  }
  return FAILURE;
} /* }}} */

static int spl_array_next_ex(spl_array_object *intern, HashTable *aht) /* {{{ */
{
  uint32_t *pos_ptr = spl_array_get_pos_ptr(aht, intern);

  zend_hash_move_forward_ex(aht, pos_ptr);
  if (spl_array_is_object(intern)) {
    return spl_array_skip_protected(intern, aht);
  } else {
    return zend_hash_has_more_elements_ex(aht, pos_ptr);
  }
} /* }}} */

static int spl_array_next(spl_array_object *intern) /* {{{ */
{
  HashTable *aht = spl_array_get_hash_table(intern);

  return spl_array_next_ex(intern, aht);

} /* }}} */

static void spl_array_it_dtor(zend_object_iterator *iter) /* {{{ */
{
  zend_user_it_invalidate_current(iter);
  zval_ptr_dtor(&iter->data);
}
/* }}} */

static int spl_array_it_valid(zend_object_iterator *iter) /* {{{ */
{
  spl_array_object *object = Z_SPLARRAY_P(&iter->data);
  HashTable *aht = spl_array_get_hash_table(object);

  if (object->ar_flags & SPL_ARRAY_OVERLOADED_VALID) {
    return zend_user_it_valid(iter);
  } else {
    if (spl_array_object_verify_pos_ex(object, aht, "ArrayIterator::valid(): ") == FAILURE) {
      return FAILURE;
    }

    return zend_hash_has_more_elements_ex(aht, spl_array_get_pos_ptr(aht, object));
  }
}
/* }}} */

static zval *spl_array_it_get_current_data(zend_object_iterator *iter) /* {{{ */
{
  spl_array_object *object = Z_SPLARRAY_P(&iter->data);
  HashTable *aht = spl_array_get_hash_table(object);

  if (object->ar_flags & SPL_ARRAY_OVERLOADED_CURRENT) {
    return zend_user_it_get_current_data(iter);
  } else {
    zval *data = zend_hash_get_current_data_ex(aht, spl_array_get_pos_ptr(aht, object));
    if (Z_TYPE_P(data) == IS_INDIRECT) {
      data = Z_INDIRECT_P(data);
    }
    return data;
  }
}
/* }}} */

static void spl_array_it_get_current_key(zend_object_iterator *iter, zval *key) /* {{{ */
{
  spl_array_object *object = Z_SPLARRAY_P(&iter->data);
  HashTable *aht = spl_array_get_hash_table(object);

  if (object->ar_flags & SPL_ARRAY_OVERLOADED_KEY) {
    zend_user_it_get_current_key(iter, key);
  } else {
    if (spl_array_object_verify_pos_ex(object, aht, "ArrayIterator::current(): ") == FAILURE) {
      ZVAL_NULL(key);
    } else {
      zend_hash_get_current_key_zval_ex(aht, key, spl_array_get_pos_ptr(aht, object));
    }
  }
}
/* }}} */

static void spl_array_it_move_forward(zend_object_iterator *iter) /* {{{ */
{
  spl_array_object *object = Z_SPLARRAY_P(&iter->data);
  HashTable *aht = spl_array_get_hash_table(object);

  if (object->ar_flags & SPL_ARRAY_OVERLOADED_NEXT) {
    zend_user_it_move_forward(iter);
  } else {
    zend_user_it_invalidate_current(iter);
    if (!aht) {
      php_error_docref(NULL, E_NOTICE, "ArrayIterator::current(): Array was modified outside object and is no longer an array");
      return;
    }

    spl_array_next_ex(object, aht);
  }
}
/* }}} */

static void spl_array_rewind(spl_array_object *intern) /* {{{ */
{
  HashTable *aht = spl_array_get_hash_table(intern);

  if (!aht) {
    php_error_docref(NULL, E_NOTICE, "ArrayIterator::rewind(): Array was modified outside object and is no longer an array");
    return;
  }

  if (intern->ht_iter == (uint32_t)-1) {
    spl_array_get_pos_ptr(aht, intern);
  } else {
    zend_hash_internal_pointer_reset_ex(aht, spl_array_get_pos_ptr(aht, intern));
    spl_array_skip_protected(intern, aht);
  }
}
/* }}} */

static void spl_array_it_rewind(zend_object_iterator *iter) /* {{{ */
{
  spl_array_object *object = Z_SPLARRAY_P(&iter->data);

  if (object->ar_flags & SPL_ARRAY_OVERLOADED_REWIND) {
    zend_user_it_rewind(iter);
  } else {
    zend_user_it_invalidate_current(iter);
    spl_array_rewind(object);
  }
}
/* }}} */

/* {{{ spl_array_set_array */
static void spl_array_set_array(zval *object, spl_array_object *intern, zval *array, zend_long ar_flags, int just_array) {
  if (Z_TYPE_P(array) != IS_OBJECT && Z_TYPE_P(array) != IS_ARRAY) {
    zend_throw_exception(spl_ce_InvalidArgumentException, "Passed variable is not an array or object", 0);
    return;
  }

  if (Z_TYPE_P(array) == IS_ARRAY) {
    zval_ptr_dtor(&intern->array);
    if (Z_REFCOUNT_P(array) == 1) {
      ZVAL_COPY(&intern->array, array);
    } else {
      //??? TODO: try to avoid array duplication
      ZVAL_ARR(&intern->array, zend_array_dup(Z_ARR_P(array)));
    }
  } else {
    if (Z_OBJ_HT_P(array) == &spl_handler_ArrayObject || Z_OBJ_HT_P(array) == &spl_handler_ArrayIterator) {
      zval_ptr_dtor(&intern->array);
      if (just_array) {
        spl_array_object *other = Z_SPLARRAY_P(array);
        ar_flags = other->ar_flags & ~SPL_ARRAY_INT_MASK;
      }
      if (Z_OBJ_P(object) == Z_OBJ_P(array)) {
        ar_flags |= SPL_ARRAY_IS_SELF;
        ZVAL_UNDEF(&intern->array);
      } else {
        ar_flags |= SPL_ARRAY_USE_OTHER;
        ZVAL_COPY(&intern->array, array);
      }
    } else {
      zend_object_get_properties_t handler = Z_OBJ_HANDLER_P(array, get_properties);
      if (handler != std_object_handlers.get_properties) {
        zend_throw_exception_ex(spl_ce_InvalidArgumentException, 0,
          "Overloaded object of type %s is not compatible with %s",
          ZSTR_VAL(Z_OBJCE_P(array)->name), ZSTR_VAL(intern->std.ce->name));
        return;
      }
      zval_ptr_dtor(&intern->array);
      ZVAL_COPY(&intern->array, array);
    }
  }

  intern->ar_flags &= ~SPL_ARRAY_IS_SELF & ~SPL_ARRAY_USE_OTHER;
  intern->ar_flags |= ar_flags;
  intern->ht_iter = (uint32_t)-1;
}
/* }}} */

/* iterator handler table */
static const zend_object_iterator_funcs spl_array_it_funcs = {
  spl_array_it_dtor,
  spl_array_it_valid,
  spl_array_it_get_current_data,
  spl_array_it_get_current_key,
  spl_array_it_move_forward,
  spl_array_it_rewind,
  NULL
};

zend_object_iterator *spl_array_get_iterator(zend_class_entry *ce, zval *object, int by_ref) /* {{{ */
{
  zend_user_iterator *iterator;
  spl_array_object *array_object = Z_SPLARRAY_P(object);

  if (by_ref && (array_object->ar_flags & SPL_ARRAY_OVERLOADED_CURRENT)) {
    zend_throw_exception(spl_ce_RuntimeException, "An iterator cannot be used with foreach by reference", 0);
    return NULL;
  }

  iterator = emalloc(sizeof(zend_user_iterator));

  zend_iterator_init(&iterator->it);

  ZVAL_COPY(&iterator->it.data, object);
  iterator->it.funcs = &spl_array_it_funcs;
  iterator->ce = ce;
  ZVAL_UNDEF(&iterator->value);

  return &iterator->it;
}
/* }}} */

/* {{{ proto void ArrayObject::__construct([array|object ar = array() [, int flags = 0 [, string iterator_class = "ArrayIterator"]]])
   Constructs a new array object from an array or object. */
SPL_METHOD(Array, __construct)
{
  zval *object = getThis();
  spl_array_object *intern;
  zval *array;
  zend_long ar_flags = 0;
  zend_class_entry *ce_get_iterator = spl_ce_Iterator;

  if (ZEND_NUM_ARGS() == 0) {
    return; /* nothing to do */
  }

  if (zend_parse_parameters_throw(ZEND_NUM_ARGS(), "z|lC", &array, &ar_flags, &ce_get_iterator) == FAILURE) {
    return;
  }

  intern = Z_SPLARRAY_P(object);

  if (ZEND_NUM_ARGS() > 2) {
    intern->ce_get_iterator = ce_get_iterator;
  }

  ar_flags &= ~SPL_ARRAY_INT_MASK;

  spl_array_set_array(object, intern, array, ar_flags, ZEND_NUM_ARGS() == 1);
}
 /* }}} */

/* {{{ proto void ArrayIterator::__construct([array|object ar = array() [, int flags = 0]])
   Constructs a new array iterator from an array or object. */
SPL_METHOD(ArrayIterator, __construct)
{
  zval *object = getThis();
  spl_array_object *intern;
  zval *array;
  zend_long ar_flags = 0;

  if (ZEND_NUM_ARGS() == 0) {
    return; /* nothing to do */
  }

  if (zend_parse_parameters_throw(ZEND_NUM_ARGS(), "z|l", &array, &ar_flags) == FAILURE) {
    return;
  }

  intern = Z_SPLARRAY_P(object);

  ar_flags &= ~SPL_ARRAY_INT_MASK;

  spl_array_set_array(object, intern, array, ar_flags, ZEND_NUM_ARGS() == 1);
}
 /* }}} */

/* {{{ proto void ArrayObject::setIteratorClass(string iterator_class)
   Set the class used in getIterator. */
SPL_METHOD(Array, setIteratorClass)
{
  zval *object = getThis();
  spl_array_object *intern = Z_SPLARRAY_P(object);
  zend_class_entry * ce_get_iterator = spl_ce_Iterator;

  ZEND_PARSE_PARAMETERS_START(1, 1)
    Z_PARAM_CLASS(ce_get_iterator)
  ZEND_PARSE_PARAMETERS_END();

  intern->ce_get_iterator = ce_get_iterator;
}
/* }}} */

/* {{{ proto string ArrayObject::getIteratorClass()
   Get the class used in getIterator. */
SPL_METHOD(Array, getIteratorClass)
{
  zval *object = getThis();
  spl_array_object *intern = Z_SPLARRAY_P(object);

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  zend_string_addref(intern->ce_get_iterator->name);
  RETURN_STR(intern->ce_get_iterator->name);
}
/* }}} */

/* {{{ proto int ArrayObject::getFlags()
   Get flags */
SPL_METHOD(Array, getFlags)
{
  zval *object = getThis();
  spl_array_object *intern = Z_SPLARRAY_P(object);

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  RETURN_LONG(intern->ar_flags & ~SPL_ARRAY_INT_MASK);
}
/* }}} */

/* {{{ proto void ArrayObject::setFlags(int flags)
   Set flags */
SPL_METHOD(Array, setFlags)
{
  zval *object = getThis();
  spl_array_object *intern = Z_SPLARRAY_P(object);
  zend_long ar_flags = 0;

  if (zend_parse_parameters(ZEND_NUM_ARGS(), "l", &ar_flags) == FAILURE) {
    return;
  }

  intern->ar_flags = (intern->ar_flags & SPL_ARRAY_INT_MASK) | (ar_flags & ~SPL_ARRAY_INT_MASK);
}
/* }}} */

/* {{{ proto Array|Object ArrayObject::exchangeArray(Array|Object ar = array())
   Replace the referenced array or object with a new one and return the old one (right now copy - to be changed) */
SPL_METHOD(Array, exchangeArray)
{
  zval *object = getThis(), *array;
  spl_array_object *intern = Z_SPLARRAY_P(object);

  if (zend_parse_parameters(ZEND_NUM_ARGS(), "z", &array) == FAILURE) {
    return;
  }

  if (intern->nApplyCount > 0) {
    zend_error(E_WARNING, "Modification of ArrayObject during sorting is prohibited");
    return;
  }

  RETVAL_ARR(zend_array_dup(spl_array_get_hash_table(intern)));
  spl_array_set_array(object, intern, array, 0L, 1);
}
/* }}} */

/* {{{ proto ArrayIterator ArrayObject::getIterator()
   Create a new iterator from a ArrayObject instance */
SPL_METHOD(Array, getIterator)
{
  zval *object = getThis();
  spl_array_object *intern = Z_SPLARRAY_P(object);
  HashTable *aht = spl_array_get_hash_table(intern);

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  if (!aht) {
    php_error_docref(NULL, E_NOTICE, "Array was modified outside object and is no longer an array");
    return;
  }

  ZVAL_OBJ(return_value, spl_array_object_new_ex(intern->ce_get_iterator, object, 0));
}
/* }}} */

/* {{{ proto void ArrayIterator::rewind()
   Rewind array back to the start */
SPL_METHOD(Array, rewind)
{
  zval *object = getThis();
  spl_array_object *intern = Z_SPLARRAY_P(object);

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  spl_array_rewind(intern);
}
/* }}} */

/* {{{ proto void ArrayIterator::seek(int $position)
   Seek to position. */
SPL_METHOD(Array, seek)
{
  zend_long opos, position;
  zval *object = getThis();
  spl_array_object *intern = Z_SPLARRAY_P(object);
  HashTable *aht = spl_array_get_hash_table(intern);
  int result;

  if (zend_parse_parameters(ZEND_NUM_ARGS(), "l", &position) == FAILURE) {
    return;
  }

  if (!aht) {
    php_error_docref(NULL, E_NOTICE, "Array was modified outside object and is no longer an array");
    return;
  }

  opos = position;

  if (position >= 0) { /* negative values are not supported */
    spl_array_rewind(intern);
    result = SUCCESS;

    while (position-- > 0 && (result = spl_array_next(intern)) == SUCCESS);

    if (result == SUCCESS && zend_hash_has_more_elements_ex(aht, spl_array_get_pos_ptr(aht, intern)) == SUCCESS) {
      return; /* ok */
    }
  }
  zend_throw_exception_ex(spl_ce_OutOfBoundsException, 0, "Seek position " ZEND_LONG_FMT " is out of range", opos);
} /* }}} */

static int spl_array_object_count_elements_helper(spl_array_object *intern, zend_long *count) /* {{{ */
{
  HashTable *aht = spl_array_get_hash_table(intern);
  HashPosition pos, *pos_ptr;

  if (!aht) {
    php_error_docref(NULL, E_NOTICE, "Array was modified outside object and is no longer an array");
    *count = 0;
    return FAILURE;
  }

  if (spl_array_is_object(intern)) {
    /* We need to store the 'pos' since we'll modify it in the functions
     * we're going to call and which do not support 'pos' as parameter. */
    pos_ptr = spl_array_get_pos_ptr(aht, intern);
    pos = *pos_ptr;
    *count = 0;
    spl_array_rewind(intern);
    while (*pos_ptr != HT_INVALID_IDX && spl_array_next(intern) == SUCCESS) {
      (*count)++;
    }
    *pos_ptr = pos;
    return SUCCESS;
  } else {
    *count = zend_hash_num_elements(aht);
    return SUCCESS;
  }
} /* }}} */

int spl_array_object_count_elements(zval *object, zend_long *count) /* {{{ */
{
  spl_array_object *intern = Z_SPLARRAY_P(object);

  if (intern->fptr_count) {
    zval rv;
    zend_call_method_with_0_params(object, intern->std.ce, &intern->fptr_count, "count", &rv);
    if (Z_TYPE(rv) != IS_UNDEF) {
      *count = zval_get_long(&rv);
      zval_ptr_dtor(&rv);
      return SUCCESS;
    }
    *count = 0;
    return FAILURE;
  }
  return spl_array_object_count_elements_helper(intern, count);
} /* }}} */

/* {{{ proto int ArrayObject::count()
       proto int ArrayIterator::count()
   Return the number of elements in the Iterator. */
SPL_METHOD(Array, count)
{
  zend_long count;
  spl_array_object *intern = Z_SPLARRAY_P(getThis());

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  spl_array_object_count_elements_helper(intern, &count);

  RETURN_LONG(count);
} /* }}} */

static void spl_array_method(INTERNAL_FUNCTION_PARAMETERS, char *fname, int fname_len, int use_arg) /* {{{ */
{
  spl_array_object *intern = Z_SPLARRAY_P(getThis());
  HashTable *aht = spl_array_get_hash_table(intern);
  zval function_name, params[2], *arg = NULL;

  ZVAL_STRINGL(&function_name, fname, fname_len);

  ZVAL_NEW_EMPTY_REF(&params[0]);
  ZVAL_ARR(Z_REFVAL(params[0]), aht);
  GC_ADDREF(aht);

  if (!use_arg) {
    intern->nApplyCount++;
    call_user_function_ex(EG(function_table), NULL, &function_name, return_value, 1, params, 1, NULL);
    intern->nApplyCount--;
  } else if (use_arg == SPL_ARRAY_METHOD_MAY_USER_ARG) {
    if (zend_parse_parameters_ex(ZEND_PARSE_PARAMS_QUIET, ZEND_NUM_ARGS(), "|z", &arg) == FAILURE) {
      zend_throw_exception(spl_ce_BadMethodCallException, "Function expects one argument at most", 0);
      goto exit;
    }
    if (arg) {
      ZVAL_COPY_VALUE(&params[1], arg);
    }
    intern->nApplyCount++;
    call_user_function_ex(EG(function_table), NULL, &function_name, return_value, arg ? 2 : 1, params, 1, NULL);
    intern->nApplyCount--;
  } else {
    if (ZEND_NUM_ARGS() != 1 || zend_parse_parameters_ex(ZEND_PARSE_PARAMS_QUIET, ZEND_NUM_ARGS(), "z", &arg) == FAILURE) {
      zend_throw_exception(spl_ce_BadMethodCallException, "Function expects exactly one argument", 0);
      goto exit;
    }
    ZVAL_COPY_VALUE(&params[1], arg);
    intern->nApplyCount++;
    call_user_function_ex(EG(function_table), NULL, &function_name, return_value, 2, params, 1, NULL);
    intern->nApplyCount--;
  }
```

```c

exit:
  {
    HashTable *new_ht = Z_ARRVAL_P(Z_REFVAL(params[0]));
    if (aht != new_ht) {
      spl_array_replace_hash_table(intern, new_ht);
    } else {
      GC_DELREF(aht);
    }
    efree(Z_REF(params[0]));
    zend_string_free(Z_STR(function_name));
  }
} /* }}} */

#define SPL_ARRAY_METHOD(cname, fname, use_arg) \
SPL_METHOD(cname, fname) \
{ \
  spl_array_method(INTERNAL_FUNCTION_PARAM_PASSTHRU, #fname, sizeof(#fname)-1, use_arg); \
}

/* {{{ proto int ArrayObject::asort([int $sort_flags = SORT_REGULAR ])
       proto int ArrayIterator::asort([int $sort_flags = SORT_REGULAR ])
   Sort the entries by values. */
SPL_ARRAY_METHOD(Array, asort, SPL_ARRAY_METHOD_MAY_USER_ARG) /* }}} */

/* {{{ proto int ArrayObject::ksort([int $sort_flags = SORT_REGULAR ])
       proto int ArrayIterator::ksort([int $sort_flags = SORT_REGULAR ])
   Sort the entries by key. */
SPL_ARRAY_METHOD(Array, ksort, SPL_ARRAY_METHOD_MAY_USER_ARG) /* }}} */

/* {{{ proto int ArrayObject::uasort(callback cmp_function)
       proto int ArrayIterator::uasort(callback cmp_function)
   Sort the entries by values user defined function. */
SPL_ARRAY_METHOD(Array, uasort, SPL_ARRAY_METHOD_USE_ARG) /* }}} */

/* {{{ proto int ArrayObject::uksort(callback cmp_function)
       proto int ArrayIterator::uksort(callback cmp_function)
   Sort the entries by key using user defined function. */
SPL_ARRAY_METHOD(Array, uksort, SPL_ARRAY_METHOD_USE_ARG) /* }}} */

/* {{{ proto int ArrayObject::natsort()
       proto int ArrayIterator::natsort()
   Sort the entries by values using "natural order" algorithm. */
SPL_ARRAY_METHOD(Array, natsort, SPL_ARRAY_METHOD_NO_ARG) /* }}} */

/* {{{ proto int ArrayObject::natcasesort()
       proto int ArrayIterator::natcasesort()
   Sort the entries by key using case insensitive "natural order" algorithm. */
SPL_ARRAY_METHOD(Array, natcasesort, SPL_ARRAY_METHOD_NO_ARG) /* }}} */

/* {{{ proto mixed|NULL ArrayIterator::current()
   Return current array entry */
SPL_METHOD(Array, current)
{
  zval *object = getThis();
  spl_array_object *intern = Z_SPLARRAY_P(object);
  zval *entry;
  HashTable *aht = spl_array_get_hash_table(intern);

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  if (spl_array_object_verify_pos(intern, aht) == FAILURE) {
    return;
  }

  if ((entry = zend_hash_get_current_data_ex(aht, spl_array_get_pos_ptr(aht, intern))) == NULL) {
    return;
  }
  if (Z_TYPE_P(entry) == IS_INDIRECT) {
    entry = Z_INDIRECT_P(entry);
    if (Z_TYPE_P(entry) == IS_UNDEF) {
      return;
    }
  }
  ZVAL_DEREF(entry);
  ZVAL_COPY(return_value, entry);
}
/* }}} */

/* {{{ proto mixed|NULL ArrayIterator::key()
   Return current array key */
SPL_METHOD(Array, key)
{
  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  spl_array_iterator_key(getThis(), return_value);
} /* }}} */

void spl_array_iterator_key(zval *object, zval *return_value) /* {{{ */
{
  spl_array_object *intern = Z_SPLARRAY_P(object);
  HashTable *aht = spl_array_get_hash_table(intern);

  if (spl_array_object_verify_pos(intern, aht) == FAILURE) {
    return;
  }

  zend_hash_get_current_key_zval_ex(aht, return_value, spl_array_get_pos_ptr(aht, intern));
}
/* }}} */

/* {{{ proto void ArrayIterator::next()
   Move to next entry */
SPL_METHOD(Array, next)
{
  zval *object = getThis();
  spl_array_object *intern = Z_SPLARRAY_P(object);
  HashTable *aht = spl_array_get_hash_table(intern);

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  if (spl_array_object_verify_pos(intern, aht) == FAILURE) {
    return;
  }

  spl_array_next_ex(intern, aht);
}
/* }}} */

/* {{{ proto bool ArrayIterator::valid()
   Check whether array contains more entries */
SPL_METHOD(Array, valid)
{
  zval *object = getThis();
  spl_array_object *intern = Z_SPLARRAY_P(object);
  HashTable *aht = spl_array_get_hash_table(intern);

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  if (spl_array_object_verify_pos(intern, aht) == FAILURE) {
    RETURN_FALSE;
  } else {
    RETURN_BOOL(zend_hash_has_more_elements_ex(aht, spl_array_get_pos_ptr(aht, intern)) == SUCCESS);
  }
}
/* }}} */

/* {{{ proto bool RecursiveArrayIterator::hasChildren()
   Check whether current element has children (e.g. is an array) */
SPL_METHOD(Array, hasChildren)
{
  zval *object = getThis(), *entry;
  spl_array_object *intern = Z_SPLARRAY_P(object);
  HashTable *aht = spl_array_get_hash_table(intern);

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  if (spl_array_object_verify_pos(intern, aht) == FAILURE) {
    RETURN_FALSE;
  }

  if ((entry = zend_hash_get_current_data_ex(aht, spl_array_get_pos_ptr(aht, intern))) == NULL) {
    RETURN_FALSE;
  }

  if (Z_TYPE_P(entry) == IS_INDIRECT) {
    entry = Z_INDIRECT_P(entry);
  }

  ZVAL_DEREF(entry);
  RETURN_BOOL(Z_TYPE_P(entry) == IS_ARRAY || (Z_TYPE_P(entry) == IS_OBJECT && (intern->ar_flags & SPL_ARRAY_CHILD_ARRAYS_ONLY) == 0));
}
/* }}} */

/* {{{ proto object RecursiveArrayIterator::getChildren()
   Create a sub iterator for the current element (same class as $this) */
SPL_METHOD(Array, getChildren)
{
  zval *object = getThis(), *entry, flags;
  spl_array_object *intern = Z_SPLARRAY_P(object);
  HashTable *aht = spl_array_get_hash_table(intern);

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  if (spl_array_object_verify_pos(intern, aht) == FAILURE) {
    return;
  }

  if ((entry = zend_hash_get_current_data_ex(aht, spl_array_get_pos_ptr(aht, intern))) == NULL) {
    return;
  }

  if (Z_TYPE_P(entry) == IS_INDIRECT) {
    entry = Z_INDIRECT_P(entry);
  }

  ZVAL_DEREF(entry);
  if (Z_TYPE_P(entry) == IS_OBJECT) {
    if ((intern->ar_flags & SPL_ARRAY_CHILD_ARRAYS_ONLY) != 0) {
      return;
    }
    if (instanceof_function(Z_OBJCE_P(entry), Z_OBJCE_P(getThis()))) {
      ZVAL_OBJ(return_value, Z_OBJ_P(entry));
      Z_ADDREF_P(return_value);
      return;
    }
  }

  ZVAL_LONG(&flags, intern->ar_flags);
  spl_instantiate_arg_ex2(Z_OBJCE_P(getThis()), return_value, entry, &flags);
}
/* }}} */

/* {{{ proto string ArrayObject::serialize()
   Serialize the object */
SPL_METHOD(Array, serialize)
{
  zval *object = getThis();
  spl_array_object *intern = Z_SPLARRAY_P(object);
  HashTable *aht = spl_array_get_hash_table(intern);
  zval members, flags;
  php_serialize_data_t var_hash;
  smart_str buf = {0};

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  if (!aht) {
    php_error_docref(NULL, E_NOTICE, "Array was modified outside object and is no longer an array");
    return;
  }

  PHP_VAR_SERIALIZE_INIT(var_hash);

  ZVAL_LONG(&flags, (intern->ar_flags & SPL_ARRAY_CLONE_MASK));

  /* storage */
  smart_str_appendl(&buf, "x:", 2);
  php_var_serialize(&buf, &flags, &var_hash);

  if (!(intern->ar_flags & SPL_ARRAY_IS_SELF)) {
    php_var_serialize(&buf, &intern->array, &var_hash);
    smart_str_appendc(&buf, ';');
  }

  /* members */
  smart_str_appendl(&buf, "m:", 2);
  if (!intern->std.properties) {
    rebuild_object_properties(&intern->std);
  }

  ZVAL_ARR(&members, intern->std.properties);

  php_var_serialize(&buf, &members, &var_hash); /* finishes the string */

  /* done */
  PHP_VAR_SERIALIZE_DESTROY(var_hash);

  if (buf.s) {
    RETURN_NEW_STR(buf.s);
  }

  RETURN_NULL();
} /* }}} */

/* {{{ proto void ArrayObject::unserialize(string serialized)
 * unserialize the object
 */
SPL_METHOD(Array, unserialize)
{
  zval *object = getThis();
  spl_array_object *intern = Z_SPLARRAY_P(object);

  char *buf;
  size_t buf_len;
  const unsigned char *p, *s;
  php_unserialize_data_t var_hash;
  zval *members, *zflags, *array;
  zend_long flags;

  if (zend_parse_parameters(ZEND_NUM_ARGS(), "s", &buf, &buf_len) == FAILURE) {
    return;
  }

  if (buf_len == 0) {
    return;
  }

  if (intern->nApplyCount > 0) {
    zend_error(E_WARNING, "Modification of ArrayObject during sorting is prohibited");
    return;
  }

  /* storage */
  s = p = (const unsigned char*)buf;
  PHP_VAR_UNSERIALIZE_INIT(var_hash);

  if (*p!= 'x' || *++p != ':') {
    goto outexcept;
  }
  ++p;

  zflags = var_tmp_var(&var_hash);
  if (!php_var_unserialize(zflags, &p, s + buf_len, &var_hash) || Z_TYPE_P(zflags) != IS_LONG) {
    goto outexcept;
  }

  --p; /* for ';' */
  flags = Z_LVAL_P(zflags);
  /* flags needs to be verified and we also need to verify whether the next
   * thing we get is ';'. After that we require an 'm' or something else
   * where 'm' stands for members and anything else should be an array. If
   * neither 'a' or 'm' follows we have an error. */

  if (*p != ';') {
    goto outexcept;
  }
  ++p;

  if (flags & SPL_ARRAY_IS_SELF) {
    /* If IS_SELF is used, the flags are not followed by an array/object */
    intern->ar_flags &= ~SPL_ARRAY_CLONE_MASK;
    intern->ar_flags |= flags & SPL_ARRAY_CLONE_MASK;
    zval_ptr_dtor(&intern->array);
    ZVAL_UNDEF(&intern->array);
  } else {
    if (*p!='a' && *p!='O' && *p!='C' && *p!='r') {
      goto outexcept;
    }

    array = var_tmp_var(&var_hash);
    if (!php_var_unserialize(array, &p, s + buf_len, &var_hash)
        || (Z_TYPE_P(array) != IS_ARRAY && Z_TYPE_P(array) != IS_OBJECT)) {
      goto outexcept;
    }

    intern->ar_flags &= ~SPL_ARRAY_CLONE_MASK;
    intern->ar_flags |= flags & SPL_ARRAY_CLONE_MASK;

    if (Z_TYPE_P(array) == IS_ARRAY) {
      zval_ptr_dtor(&intern->array);
      ZVAL_COPY(&intern->array, array);
    } else {
      spl_array_set_array(object, intern, array, 0L, 1);
    }

    if (*p != ';') {
      goto outexcept;
    }
        ++p;
  }

  /* members */
  if (*p!= 'm' || *++p != ':') {
    goto outexcept;
  }
  ++p;

  members = var_tmp_var(&var_hash);
  if (!php_var_unserialize(members, &p, s + buf_len, &var_hash) || Z_TYPE_P(members) != IS_ARRAY) {
    goto outexcept;
  }

  /* copy members */
  object_properties_load(&intern->std, Z_ARRVAL_P(members));

  /* done reading $serialized */
  PHP_VAR_UNSERIALIZE_DESTROY(var_hash);
  return;

outexcept:
  PHP_VAR_UNSERIALIZE_DESTROY(var_hash);
  zend_throw_exception_ex(spl_ce_UnexpectedValueException, 0, "Error at offset " ZEND_LONG_FMT " of %zd bytes", (zend_long)((char*)p - buf), buf_len);
  return;

} /* }}} */

/* {{{ arginfo and function table */
ZEND_BEGIN_ARG_INFO_EX(arginfo_array___construct, 0, 0, 0)
  ZEND_ARG_INFO(0, array)
  ZEND_ARG_INFO(0, ar_flags)
  ZEND_ARG_INFO(0, iterator_class)
ZEND_END_ARG_INFO()

/* ArrayIterator::__construct and ArrayObject::__construct have different signatures */
ZEND_BEGIN_ARG_INFO_EX(arginfo_array_iterator___construct, 0, 0, 0)
  ZEND_ARG_INFO(0, array)
  ZEND_ARG_INFO(0, ar_flags)
ZEND_END_ARG_INFO()

ZEND_BEGIN_ARG_INFO_EX(arginfo_array_offsetGet, 0, 0, 1)
  ZEND_ARG_INFO(0, index)
ZEND_END_ARG_INFO()

ZEND_BEGIN_ARG_INFO_EX(arginfo_array_offsetSet, 0, 0, 2)
  ZEND_ARG_INFO(0, index)
  ZEND_ARG_INFO(0, newval)
ZEND_END_ARG_INFO()

ZEND_BEGIN_ARG_INFO(arginfo_array_append, 0)
  ZEND_ARG_INFO(0, value)
ZEND_END_ARG_INFO()

ZEND_BEGIN_ARG_INFO(arginfo_array_seek, 0)
  ZEND_ARG_INFO(0, position)
ZEND_END_ARG_INFO()

ZEND_BEGIN_ARG_INFO(arginfo_array_exchangeArray, 0)
  ZEND_ARG_INFO(0, array)
ZEND_END_ARG_INFO()

ZEND_BEGIN_ARG_INFO(arginfo_array_setFlags, 0)
  ZEND_ARG_INFO(0, flags)
ZEND_END_ARG_INFO()

ZEND_BEGIN_ARG_INFO(arginfo_array_setIteratorClass, 0)
  ZEND_ARG_INFO(0, iteratorClass)
ZEND_END_ARG_INFO()

ZEND_BEGIN_ARG_INFO(arginfo_array_uXsort, 0)
  ZEND_ARG_INFO(0, cmp_function)
ZEND_END_ARG_INFO();

ZEND_BEGIN_ARG_INFO(arginfo_array_unserialize, 0)
  ZEND_ARG_INFO(0, serialized)
ZEND_END_ARG_INFO();

ZEND_BEGIN_ARG_INFO(arginfo_array_void, 0)
ZEND_END_ARG_INFO()

static const zend_function_entry spl_funcs_ArrayObject[] = {
  SPL_ME(Array, __construct,      arginfo_array___construct,      ZEND_ACC_PUBLIC)
  SPL_ME(Array, offsetExists,     arginfo_array_offsetGet,        ZEND_ACC_PUBLIC)
  SPL_ME(Array, offsetGet,        arginfo_array_offsetGet,        ZEND_ACC_PUBLIC)
  SPL_ME(Array, offsetSet,        arginfo_array_offsetSet,        ZEND_ACC_PUBLIC)
  SPL_ME(Array, offsetUnset,      arginfo_array_offsetGet,        ZEND_ACC_PUBLIC)
  SPL_ME(Array, append,           arginfo_array_append,           ZEND_ACC_PUBLIC)
  SPL_ME(Array, getArrayCopy,     arginfo_array_void,             ZEND_ACC_PUBLIC)
  SPL_ME(Array, count,            arginfo_array_void,             ZEND_ACC_PUBLIC)
  SPL_ME(Array, getFlags,         arginfo_array_void,             ZEND_ACC_PUBLIC)
  SPL_ME(Array, setFlags,         arginfo_array_setFlags,         ZEND_ACC_PUBLIC)
  SPL_ME(Array, asort,            arginfo_array_void,             ZEND_ACC_PUBLIC)
  SPL_ME(Array, ksort,            arginfo_array_void,             ZEND_ACC_PUBLIC)
  SPL_ME(Array, uasort,           arginfo_array_uXsort,           ZEND_ACC_PUBLIC)
  SPL_ME(Array, uksort,           arginfo_array_uXsort,           ZEND_ACC_PUBLIC)
  SPL_ME(Array, natsort,          arginfo_array_void,             ZEND_ACC_PUBLIC)
  SPL_ME(Array, natcasesort,      arginfo_array_void,             ZEND_ACC_PUBLIC)
  SPL_ME(Array, unserialize,      arginfo_array_unserialize,      ZEND_ACC_PUBLIC)
  SPL_ME(Array, serialize,        arginfo_array_void,             ZEND_ACC_PUBLIC)
  /* ArrayObject specific */
  SPL_ME(Array, getIterator,      arginfo_array_void,             ZEND_ACC_PUBLIC)
  SPL_ME(Array, exchangeArray,    arginfo_array_exchangeArray,    ZEND_ACC_PUBLIC)
  SPL_ME(Array, setIteratorClass, arginfo_array_setIteratorClass, ZEND_ACC_PUBLIC)
  SPL_ME(Array, getIteratorClass, arginfo_array_void,             ZEND_ACC_PUBLIC)
  PHP_FE_END
};

static const zend_function_entry spl_funcs_ArrayIterator[] = {
  SPL_ME(ArrayIterator, __construct, arginfo_array_iterator___construct,      ZEND_ACC_PUBLIC)
  SPL_ME(Array, offsetExists,     arginfo_array_offsetGet,        ZEND_ACC_PUBLIC)
  SPL_ME(Array, offsetGet,        arginfo_array_offsetGet,        ZEND_ACC_PUBLIC)
  SPL_ME(Array, offsetSet,        arginfo_array_offsetSet,        ZEND_ACC_PUBLIC)
  SPL_ME(Array, offsetUnset,      arginfo_array_offsetGet,        ZEND_ACC_PUBLIC)
  SPL_ME(Array, append,           arginfo_array_append,           ZEND_ACC_PUBLIC)
  SPL_ME(Array, getArrayCopy,     arginfo_array_void,             ZEND_ACC_PUBLIC)
  SPL_ME(Array, count,            arginfo_array_void,             ZEND_ACC_PUBLIC)
  SPL_ME(Array, getFlags,         arginfo_array_void,             ZEND_ACC_PUBLIC)
  SPL_ME(Array, setFlags,         arginfo_array_setFlags,         ZEND_ACC_PUBLIC)
  SPL_ME(Array, asort,            arginfo_array_void,             ZEND_ACC_PUBLIC)
  SPL_ME(Array, ksort,            arginfo_array_void,             ZEND_ACC_PUBLIC)
  SPL_ME(Array, uasort,           arginfo_array_uXsort,           ZEND_ACC_PUBLIC)
  SPL_ME(Array, uksort,           arginfo_array_uXsort,           ZEND_ACC_PUBLIC)
  SPL_ME(Array, natsort,          arginfo_array_void,             ZEND_ACC_PUBLIC)
  SPL_ME(Array, natcasesort,      arginfo_array_void,             ZEND_ACC_PUBLIC)
  SPL_ME(Array, unserialize,      arginfo_array_unserialize,      ZEND_ACC_PUBLIC)
  SPL_ME(Array, serialize,        arginfo_array_void,             ZEND_ACC_PUBLIC)
  /* ArrayIterator specific */
  SPL_ME(Array, rewind,           arginfo_array_void,             ZEND_ACC_PUBLIC)
  SPL_ME(Array, current,          arginfo_array_void,             ZEND_ACC_PUBLIC)
  SPL_ME(Array, key,              arginfo_array_void,             ZEND_ACC_PUBLIC)
  SPL_ME(Array, next,             arginfo_array_void,             ZEND_ACC_PUBLIC)
  SPL_ME(Array, valid,            arginfo_array_void,             ZEND_ACC_PUBLIC)
  SPL_ME(Array, seek,             arginfo_array_seek,             ZEND_ACC_PUBLIC)
  PHP_FE_END
};

static const zend_function_entry spl_funcs_RecursiveArrayIterator[] = {
  SPL_ME(Array, hasChildren,   arginfo_array_void, ZEND_ACC_PUBLIC)
  SPL_ME(Array, getChildren,   arginfo_array_void, ZEND_ACC_PUBLIC)
  PHP_FE_END
};
/* }}} */

/* {{{ PHP_MINIT_FUNCTION(spl_array) */
PHP_MINIT_FUNCTION(spl_array)
{
  REGISTER_SPL_STD_CLASS_EX(ArrayObject, spl_array_object_new, spl_funcs_ArrayObject);
  REGISTER_SPL_IMPLEMENTS(ArrayObject, Aggregate);
  REGISTER_SPL_IMPLEMENTS(ArrayObject, ArrayAccess);
  REGISTER_SPL_IMPLEMENTS(ArrayObject, Serializable);
  REGISTER_SPL_IMPLEMENTS(ArrayObject, Countable);
  memcpy(&spl_handler_ArrayObject, zend_get_std_object_handlers(), sizeof(zend_object_handlers));

  spl_handler_ArrayObject.offset = XtOffsetOf(spl_array_object, std);

  spl_handler_ArrayObject.clone_obj = spl_array_object_clone;
  spl_handler_ArrayObject.read_dimension = spl_array_read_dimension;
  spl_handler_ArrayObject.write_dimension = spl_array_write_dimension;
  spl_handler_ArrayObject.unset_dimension = spl_array_unset_dimension;
  spl_handler_ArrayObject.has_dimension = spl_array_has_dimension;
  spl_handler_ArrayObject.count_elements = spl_array_object_count_elements;

  spl_handler_ArrayObject.get_properties = spl_array_get_properties;
  spl_handler_ArrayObject.get_debug_info = spl_array_get_debug_info;
  spl_handler_ArrayObject.get_gc = spl_array_get_gc;
  spl_handler_ArrayObject.read_property = spl_array_read_property;
  spl_handler_ArrayObject.write_property = spl_array_write_property;
  spl_handler_ArrayObject.get_property_ptr_ptr = spl_array_get_property_ptr_ptr;
  spl_handler_ArrayObject.has_property = spl_array_has_property;
  spl_handler_ArrayObject.unset_property = spl_array_unset_property;

  spl_handler_ArrayObject.compare_objects = spl_array_compare_objects;
  spl_handler_ArrayObject.dtor_obj = zend_objects_destroy_object;
  spl_handler_ArrayObject.free_obj = spl_array_object_free_storage;

  REGISTER_SPL_STD_CLASS_EX(ArrayIterator, spl_array_object_new, spl_funcs_ArrayIterator);
  REGISTER_SPL_IMPLEMENTS(ArrayIterator, Iterator);
  REGISTER_SPL_IMPLEMENTS(ArrayIterator, ArrayAccess);
  REGISTER_SPL_IMPLEMENTS(ArrayIterator, SeekableIterator);
  REGISTER_SPL_IMPLEMENTS(ArrayIterator, Serializable);
  REGISTER_SPL_IMPLEMENTS(ArrayIterator, Countable);
  memcpy(&spl_handler_ArrayIterator, &spl_handler_ArrayObject, sizeof(zend_object_handlers));
  spl_ce_ArrayIterator->get_iterator = spl_array_get_iterator;

  REGISTER_SPL_CLASS_CONST_LONG(ArrayObject,   "STD_PROP_LIST",    SPL_ARRAY_STD_PROP_LIST);
  REGISTER_SPL_CLASS_CONST_LONG(ArrayObject,   "ARRAY_AS_PROPS",   SPL_ARRAY_ARRAY_AS_PROPS);

  REGISTER_SPL_CLASS_CONST_LONG(ArrayIterator, "STD_PROP_LIST",    SPL_ARRAY_STD_PROP_LIST);
  REGISTER_SPL_CLASS_CONST_LONG(ArrayIterator, "ARRAY_AS_PROPS",   SPL_ARRAY_ARRAY_AS_PROPS);

  REGISTER_SPL_SUB_CLASS_EX(RecursiveArrayIterator, ArrayIterator, spl_array_object_new, spl_funcs_RecursiveArrayIterator);
  REGISTER_SPL_IMPLEMENTS(RecursiveArrayIterator, RecursiveIterator);
  spl_ce_RecursiveArrayIterator->get_iterator = spl_array_get_iterator;

  REGISTER_SPL_CLASS_CONST_LONG(RecursiveArrayIterator, "CHILD_ARRAYS_ONLY", SPL_ARRAY_CHILD_ARRAYS_ONLY);

  return SUCCESS;
}
/* }}} */

/*
 * Local variables:
 * tab-width: 4
 * c-basic-offset: 4
 * End:
 * vim600: fdm=marker
 * vim: noet sw=4 ts=4
 */
```

```c
/*
   +----------------------------------------------------------------------+
   | PHP Version 7                                                        |
   +----------------------------------------------------------------------+
   | Copyright (c) 1997-2018 The PHP Group                                |
   +----------------------------------------------------------------------+
   | This source file is subject to version 3.01 of the PHP license,      |
   | that is bundled with this package in the file LICENSE, and is        |
   | available through the world-wide-web at the following url:           |
   | http://www.php.net/license/3_01.txt                                  |
   | If you did not receive a copy of the PHP license and are unable to   |
   | obtain it through the world-wide-web, please send a note to          |
   | license@php.net so we can mail you a copy immediately.               |
   +----------------------------------------------------------------------+
   | Authors: Etienne Kneuss <colder@php.net>                             |
   +----------------------------------------------------------------------+
 */

/* $Id$ */

#ifndef SPL_HEAP_H
#define SPL_HEAP_H

#include "php.h"
#include "php_spl.h"

extern PHPAPI zend_class_entry *spl_ce_SplHeap;
extern PHPAPI zend_class_entry *spl_ce_SplMinHeap;
extern PHPAPI zend_class_entry *spl_ce_SplMaxHeap;

extern PHPAPI zend_class_entry *spl_ce_SplPriorityQueue;

PHP_MINIT_FUNCTION(spl_heap);

#endif /* SPL_HEAP_H */

/*
 * Local Variables:
 * c-basic-offset: 4
 * tab-width: 4
 * End:
 * vim600: fdm=marker
 * vim: noet sw=4 ts=4
 */
```

```c
/*
   +----------------------------------------------------------------------+
   | PHP Version 7                                                        |
   +----------------------------------------------------------------------+
   | Copyright (c) 1997-2018 The PHP Group                                |
   +----------------------------------------------------------------------+
   | This source file is subject to version 3.01 of the PHP license,      |
   | that is bundled with this package in the file LICENSE, and is        |
   | available through the world-wide-web at the following url:           |
   | http://www.php.net/license/3_01.txt                                  |
   | If you did not receive a copy of the PHP license and are unable to   |
   | obtain it through the world-wide-web, please send a note to          |
   | license@php.net so we can mail you a copy immediately.               |
   +----------------------------------------------------------------------+
   | Authors: Marcus Boerger <helly@php.net>                              |
   +----------------------------------------------------------------------+
 */

/* $Id$ */

#ifndef SPL_ENGINE_H
#define SPL_ENGINE_H

#include "php.h"
#include "php_spl.h"
#include "zend_interfaces.h"

PHPAPI void spl_instantiate(zend_class_entry *pce, zval *object);

PHPAPI zend_long spl_offset_convert_to_long(zval *offset);

/* {{{ spl_instantiate_arg_ex1 */
static inline int spl_instantiate_arg_ex1(zend_class_entry *pce, zval *retval, zval *arg1)
{
  zend_function *func = pce->constructor;
  spl_instantiate(pce, retval);

  zend_call_method(retval, pce, &func, ZSTR_VAL(func->common.function_name), ZSTR_LEN(func->common.function_name), NULL, 1, arg1, NULL);
  return 0;
}
/* }}} */

/* {{{ spl_instantiate_arg_ex2 */
static inline int spl_instantiate_arg_ex2(zend_class_entry *pce, zval *retval, zval *arg1, zval *arg2)
{
  zend_function *func = pce->constructor;
  spl_instantiate(pce, retval);

  zend_call_method(retval, pce, &func, ZSTR_VAL(func->common.function_name), ZSTR_LEN(func->common.function_name), NULL, 2, arg1, arg2);
  return 0;
}
/* }}} */

/* {{{ spl_instantiate_arg_n */
static inline void spl_instantiate_arg_n(zend_class_entry *pce, zval *retval, int argc, zval *argv)
{
  zend_function *func = pce->constructor;
  zend_fcall_info fci;
  zend_fcall_info_cache fcc;
  zval dummy;

  spl_instantiate(pce, retval);

  fci.size = sizeof(zend_fcall_info);
  ZVAL_STR(&fci.function_name, func->common.function_name);
  fci.object = Z_OBJ_P(retval);
  fci.retval = &dummy;
  fci.param_count = argc;
  fci.params = argv;
  fci.no_separation = 1;

  fcc.function_handler = func;
  fcc.calling_scope = zend_get_executed_scope();
  fcc.called_scope = pce;
  fcc.object = Z_OBJ_P(retval);

  zend_call_function(&fci, &fcc);
}
/* }}} */

#endif /* SPL_ENGINE_H */

/*
 * Local Variables:
 * c-basic-offset: 4
 * tab-width: 4
 * End:
 * vim600: fdm=marker
 * vim: noet sw=4 ts=4
 */
```

```c
/*
   +----------------------------------------------------------------------+
   | PHP Version 7                                                        |
   +----------------------------------------------------------------------+
   | Copyright (c) 1997-2018 The PHP Group                                |
   +----------------------------------------------------------------------+
   | This source file is subject to version 3.01 of the PHP license,      |
   | that is bundled with this package in the file LICENSE, and is        |
   | available through the world-wide-web at the following url:           |
   | http://www.php.net/license/3_01.txt                                  |
   | If you did not receive a copy of the PHP license and are unable to   |
   | obtain it through the world-wide-web, please send a note to          |
   | license@php.net so we can mail you a copy immediately.               |
   +----------------------------------------------------------------------+
   | Authors: Marcus Boerger <helly@php.net>                              |
   +----------------------------------------------------------------------+
 */

/* $Id$ */

#ifdef HAVE_CONFIG_H
# include "config.h"
#endif

#include "php.h"
#include "php_ini.h"
#include "ext/standard/info.h"
#include "zend_exceptions.h"
#include "zend_interfaces.h"

#include "php_spl.h"
#include "spl_functions.h"
#include "spl_engine.h"
#include "spl_iterators.h"
#include "spl_directory.h"
#include "spl_array.h"
#include "spl_exceptions.h"
#include "zend_smart_str.h"

#ifdef accept
#undef accept
#endif

PHPAPI zend_class_entry *spl_ce_RecursiveIterator;
PHPAPI zend_class_entry *spl_ce_RecursiveIteratorIterator;
PHPAPI zend_class_entry *spl_ce_FilterIterator;
PHPAPI zend_class_entry *spl_ce_CallbackFilterIterator;
PHPAPI zend_class_entry *spl_ce_RecursiveFilterIterator;
PHPAPI zend_class_entry *spl_ce_RecursiveCallbackFilterIterator;
PHPAPI zend_class_entry *spl_ce_ParentIterator;
PHPAPI zend_class_entry *spl_ce_SeekableIterator;
PHPAPI zend_class_entry *spl_ce_LimitIterator;
PHPAPI zend_class_entry *spl_ce_CachingIterator;
PHPAPI zend_class_entry *spl_ce_RecursiveCachingIterator;
PHPAPI zend_class_entry *spl_ce_OuterIterator;
PHPAPI zend_class_entry *spl_ce_IteratorIterator;
PHPAPI zend_class_entry *spl_ce_NoRewindIterator;
PHPAPI zend_class_entry *spl_ce_InfiniteIterator;
PHPAPI zend_class_entry *spl_ce_EmptyIterator;
PHPAPI zend_class_entry *spl_ce_AppendIterator;
PHPAPI zend_class_entry *spl_ce_RegexIterator;
PHPAPI zend_class_entry *spl_ce_RecursiveRegexIterator;
PHPAPI zend_class_entry *spl_ce_RecursiveTreeIterator;

ZEND_BEGIN_ARG_INFO(arginfo_recursive_it_void, 0)
ZEND_END_ARG_INFO()

static const zend_function_entry spl_funcs_RecursiveIterator[] = {
  SPL_ABSTRACT_ME(RecursiveIterator, hasChildren,  arginfo_recursive_it_void)
  SPL_ABSTRACT_ME(RecursiveIterator, getChildren,  arginfo_recursive_it_void)
  PHP_FE_END
};

typedef enum {
  RIT_LEAVES_ONLY = 0,
  RIT_SELF_FIRST  = 1,
  RIT_CHILD_FIRST = 2
} RecursiveIteratorMode;

#define RIT_CATCH_GET_CHILD CIT_CATCH_GET_CHILD

typedef enum {
  RTIT_BYPASS_CURRENT = 4,
  RTIT_BYPASS_KEY     = 8
} RecursiveTreeIteratorFlags;

typedef enum {
  RS_NEXT  = 0,
  RS_TEST  = 1,
  RS_SELF  = 2,
  RS_CHILD = 3,
  RS_START = 4
} RecursiveIteratorState;

typedef struct _spl_sub_iterator {
  zend_object_iterator    *iterator;
  zval                    zobject;
  zend_class_entry        *ce;
  RecursiveIteratorState  state;
} spl_sub_iterator;

typedef struct _spl_recursive_it_object {
  spl_sub_iterator         *iterators;
  int                      level;
  RecursiveIteratorMode    mode;
  int                      flags;
  int                      max_depth;
  zend_bool                in_iteration;
  zend_function            *beginIteration;
  zend_function            *endIteration;
  zend_function            *callHasChildren;
  zend_function            *callGetChildren;
  zend_function            *beginChildren;
  zend_function            *endChildren;
  zend_function            *nextElement;
  zend_class_entry         *ce;
  smart_str                prefix[6];
  smart_str                postfix[1];
  zend_object              std;
} spl_recursive_it_object;

typedef struct _spl_recursive_it_iterator {
  zend_object_iterator   intern;
} spl_recursive_it_iterator;

static zend_object_handlers spl_handlers_rec_it_it;
static zend_object_handlers spl_handlers_dual_it;

static inline spl_recursive_it_object *spl_recursive_it_from_obj(zend_object *obj) /* {{{ */ {
  return (spl_recursive_it_object*)((char*)(obj) - XtOffsetOf(spl_recursive_it_object, std));
}
/* }}} */

#define Z_SPLRECURSIVE_IT_P(zv)  spl_recursive_it_from_obj(Z_OBJ_P((zv)))

#define SPL_FETCH_AND_CHECK_DUAL_IT(var, objzval)                          \
  do {                                                     \
    spl_dual_it_object *it = Z_SPLDUAL_IT_P(objzval);                      \
    if (it->dit_type == DIT_Unknown) {                             \
      zend_throw_exception_ex(spl_ce_LogicException, 0,              \
        "The object is in an invalid state as the parent constructor was not called");  \
      return;                                                \
    }                                                      \
    (var) = it;                                                \
  } while (0)

#define SPL_FETCH_SUB_ELEMENT(var, object, element) \
  do { \
    if(!(object)->iterators) { \
      zend_throw_exception_ex(spl_ce_LogicException, 0, \
        "The object is in an invalid state as the parent constructor was not called"); \
      return; \
    } \
    (var) = (object)->iterators[(object)->level].element; \
  } while (0)

#define SPL_FETCH_SUB_ELEMENT_ADDR(var, object, element) \
  do { \
    if(!(object)->iterators) { \
      zend_throw_exception_ex(spl_ce_LogicException, 0, \
        "The object is in an invalid state as the parent constructor was not called"); \
      return; \
    } \
    (var) = &(object)->iterators[(object)->level].element; \
  } while (0)

#define SPL_FETCH_SUB_ITERATOR(var, object) SPL_FETCH_SUB_ELEMENT(var, object, iterator)


static void spl_recursive_it_dtor(zend_object_iterator *_iter)
{
  spl_recursive_it_iterator *iter   = (spl_recursive_it_iterator*)_iter;
  spl_recursive_it_object   *object = Z_SPLRECURSIVE_IT_P(&iter->intern.data);
  zend_object_iterator      *sub_iter;

  while (object->level > 0) {
    if (!Z_ISUNDEF(object->iterators[object->level].zobject)) {
      sub_iter = object->iterators[object->level].iterator;
      zend_iterator_dtor(sub_iter);
      zval_ptr_dtor(&object->iterators[object->level].zobject);
    }
    object->level--;
  }
  object->iterators = erealloc(object->iterators, sizeof(spl_sub_iterator));
  object->level = 0;

  zval_ptr_dtor(&iter->intern.data);
}

static int spl_recursive_it_valid_ex(spl_recursive_it_object *object, zval *zthis)
{
  zend_object_iterator      *sub_iter;
  int                       level = object->level;

  if(!object->iterators) {
    return FAILURE;
  }
  while (level >=0) {
    sub_iter = object->iterators[level].iterator;
    if (sub_iter->funcs->valid(sub_iter) == SUCCESS) {
      return SUCCESS;
    }
    level--;
  }
  if (object->endIteration && object->in_iteration) {
    zend_call_method_with_0_params(zthis, object->ce, &object->endIteration, "endIteration", NULL);
  }
  object->in_iteration = 0;
  return FAILURE;
}

static int spl_recursive_it_valid(zend_object_iterator *iter)
{
  return spl_recursive_it_valid_ex(Z_SPLRECURSIVE_IT_P(&iter->data), &iter->data);
}

static zval *spl_recursive_it_get_current_data(zend_object_iterator *iter)
{
  spl_recursive_it_object *object = Z_SPLRECURSIVE_IT_P(&iter->data);
  zend_object_iterator *sub_iter = object->iterators[object->level].iterator;

  return sub_iter->funcs->get_current_data(sub_iter);
}

static void spl_recursive_it_get_current_key(zend_object_iterator *iter, zval *key)
{
  spl_recursive_it_object *object = Z_SPLRECURSIVE_IT_P(&iter->data);
  zend_object_iterator *sub_iter = object->iterators[object->level].iterator;

  if (sub_iter->funcs->get_current_key) {
    sub_iter->funcs->get_current_key(sub_iter, key);
  } else {
    ZVAL_LONG(key, iter->index);
  }
}

static void spl_recursive_it_move_forward_ex(spl_recursive_it_object *object, zval *zthis)
{
  zend_object_iterator      *iterator;
  zval                      *zobject;
  zend_class_entry          *ce;
  zval                      retval, child;
  zend_object_iterator      *sub_iter;
  int                       has_children;

  SPL_FETCH_SUB_ITERATOR(iterator, object);

  while (!EG(exception)) {
next_step:
    iterator = object->iterators[object->level].iterator;
    switch (object->iterators[object->level].state) {
      case RS_NEXT:
        iterator->funcs->move_forward(iterator);
        if (EG(exception)) {
          if (!(object->flags & RIT_CATCH_GET_CHILD)) {
            return;
          } else {
            zend_clear_exception();
          }
        }
        /* fall through */
      case RS_START:
        if (iterator->funcs->valid(iterator) == FAILURE) {
          break;
        }
        object->iterators[object->level].state = RS_TEST;
        /* break; */
      case RS_TEST:
        ce = object->iterators[object->level].ce;
        zobject = &object->iterators[object->level].zobject;
        if (object->callHasChildren) {
          zend_call_method_with_0_params(zthis, object->ce, &object->callHasChildren, "callHasChildren", &retval);
        } else {
          zend_call_method_with_0_params(zobject, ce, NULL, "haschildren", &retval);
        }
        if (EG(exception)) {
          if (!(object->flags & RIT_CATCH_GET_CHILD)) {
            object->iterators[object->level].state = RS_NEXT;
            return;
          } else {
            zend_clear_exception();
          }
        }
        if (Z_TYPE(retval) != IS_UNDEF) {
          has_children = zend_is_true(&retval);
          zval_ptr_dtor(&retval);
          if (has_children) {
            if (object->max_depth == -1 || object->max_depth > object->level) {
              switch (object->mode) {
              case RIT_LEAVES_ONLY:
              case RIT_CHILD_FIRST:
                object->iterators[object->level].state = RS_CHILD;
                goto next_step;
              case RIT_SELF_FIRST:
                object->iterators[object->level].state = RS_SELF;
                goto next_step;
              }
            } else {
              /* do not recurse into */
              if (object->mode == RIT_LEAVES_ONLY) {
                /* this is not a leave, so skip it */
                object->iterators[object->level].state = RS_NEXT;
                goto next_step;
              }
            }
          }
        }
        if (object->nextElement) {
          zend_call_method_with_0_params(zthis, object->ce, &object->nextElement, "nextelement", NULL);
        }
        object->iterators[object->level].state = RS_NEXT;
        if (EG(exception)) {
          if (!(object->flags & RIT_CATCH_GET_CHILD)) {
            return;
          } else {
            zend_clear_exception();
          }
        }
        return /* self */;
      case RS_SELF:
        if (object->nextElement && (object->mode == RIT_SELF_FIRST || object->mode == RIT_CHILD_FIRST)) {
          zend_call_method_with_0_params(zthis, object->ce, &object->nextElement, "nextelement", NULL);
        }
        if (object->mode == RIT_SELF_FIRST) {
          object->iterators[object->level].state = RS_CHILD;
        } else {
          object->iterators[object->level].state = RS_NEXT;
        }
        return /* self */;
      case RS_CHILD:
        ce = object->iterators[object->level].ce;
        zobject = &object->iterators[object->level].zobject;
        if (object->callGetChildren) {
          zend_call_method_with_0_params(zthis, object->ce, &object->callGetChildren, "callGetChildren", &child);
        } else {
          zend_call_method_with_0_params(zobject, ce, NULL, "getchildren", &child);
        }

        if (EG(exception)) {
          if (!(object->flags & RIT_CATCH_GET_CHILD)) {
            return;
          } else {
            zend_clear_exception();
            zval_ptr_dtor(&child);
            object->iterators[object->level].state = RS_NEXT;
            goto next_step;
          }
        }

        if (Z_TYPE(child) == IS_UNDEF || Z_TYPE(child) != IS_OBJECT ||
            !((ce = Z_OBJCE(child)) && instanceof_function(ce, spl_ce_RecursiveIterator))) {
          zval_ptr_dtor(&child);
          zend_throw_exception(spl_ce_UnexpectedValueException, "Objects returned by RecursiveIterator::getChildren() must implement RecursiveIterator", 0);
          return;
        }

        if (object->mode == RIT_CHILD_FIRST) {
          object->iterators[object->level].state = RS_SELF;
        } else {
          object->iterators[object->level].state = RS_NEXT;
        }
        object->iterators = erealloc(object->iterators, sizeof(spl_sub_iterator) * (++object->level+1));
        sub_iter = ce->get_iterator(ce, &child, 0);
        ZVAL_COPY_VALUE(&object->iterators[object->level].zobject, &child);
        object->iterators[object->level].iterator = sub_iter;
        object->iterators[object->level].ce = ce;
        object->iterators[object->level].state = RS_START;
        if (sub_iter->funcs->rewind) {
          sub_iter->funcs->rewind(sub_iter);
        }
        if (object->beginChildren) {
          zend_call_method_with_0_params(zthis, object->ce, &object->beginChildren, "beginchildren", NULL);
          if (EG(exception)) {
            if (!(object->flags & RIT_CATCH_GET_CHILD)) {
              return;
            } else {
              zend_clear_exception();
            }
          }
        }
        goto next_step;
    }
    /* no more elements */
    if (object->level > 0) {
      if (object->endChildren) {
        zend_call_method_with_0_params(zthis, object->ce, &object->endChildren, "endchildren", NULL);
        if (EG(exception)) {
          if (!(object->flags & RIT_CATCH_GET_CHILD)) {
            return;
          } else {
            zend_clear_exception();
          }
        }
      }
      if (object->level > 0) {
        zval garbage;
        ZVAL_COPY_VALUE(&garbage, &object->iterators[object->level].zobject);
        ZVAL_UNDEF(&object->iterators[object->level].zobject);
        zval_ptr_dtor(&garbage);
        zend_iterator_dtor(iterator);
        object->level--;
      }
    } else {
      return; /* done completeley */
    }
  }
}

static void spl_recursive_it_rewind_ex(spl_recursive_it_object *object, zval *zthis)
{
  zend_object_iterator *sub_iter;

  SPL_FETCH_SUB_ITERATOR(sub_iter, object);

  while (object->level) {
    sub_iter = object->iterators[object->level].iterator;
    zend_iterator_dtor(sub_iter);
    zval_ptr_dtor(&object->iterators[object->level--].zobject);
    if (!EG(exception) && (!object->endChildren || object->endChildren->common.scope != spl_ce_RecursiveIteratorIterator)) {
      zend_call_method_with_0_params(zthis, object->ce, &object->endChildren, "endchildren", NULL);
    }
  }
  object->iterators = erealloc(object->iterators, sizeof(spl_sub_iterator));
  object->iterators[0].state = RS_START;
  sub_iter = object->iterators[0].iterator;
  if (sub_iter->funcs->rewind) {
    sub_iter->funcs->rewind(sub_iter);
  }
  if (!EG(exception) && object->beginIteration && !object->in_iteration) {
    zend_call_method_with_0_params(zthis, object->ce, &object->beginIteration, "beginIteration", NULL);
  }
  object->in_iteration = 1;
  spl_recursive_it_move_forward_ex(object, zthis);
}

static void spl_recursive_it_move_forward(zend_object_iterator *iter)
{
  spl_recursive_it_move_forward_ex(Z_SPLRECURSIVE_IT_P(&iter->data), &iter->data);
}

static void spl_recursive_it_rewind(zend_object_iterator *iter)
{
  spl_recursive_it_rewind_ex(Z_SPLRECURSIVE_IT_P(&iter->data), &iter->data);
}

static zend_object_iterator *spl_recursive_it_get_iterator(zend_class_entry *ce, zval *zobject, int by_ref)
{
  spl_recursive_it_iterator *iterator;
  spl_recursive_it_object *object;

  if (by_ref) {
    zend_throw_exception(spl_ce_RuntimeException, "An iterator cannot be used with foreach by reference", 0);
    return NULL;
  }
  iterator = emalloc(sizeof(spl_recursive_it_iterator));
  object   = Z_SPLRECURSIVE_IT_P(zobject);
  if (object->iterators == NULL) {
    zend_error(E_ERROR, "The object to be iterated is in an invalid state: "
        "the parent constructor has not been called");
  }

  zend_iterator_init((zend_object_iterator*)iterator);

  ZVAL_COPY(&iterator->intern.data, zobject);
  iterator->intern.funcs = ce->iterator_funcs.funcs;
  return (zend_object_iterator*)iterator;
}

static const zend_object_iterator_funcs spl_recursive_it_iterator_funcs = {
  spl_recursive_it_dtor,
  spl_recursive_it_valid,
  spl_recursive_it_get_current_data,
  spl_recursive_it_get_current_key,
  spl_recursive_it_move_forward,
  spl_recursive_it_rewind,
  NULL
};

static void spl_recursive_it_it_construct(INTERNAL_FUNCTION_PARAMETERS, zend_class_entry *ce_base, zend_class_entry *ce_inner, recursive_it_it_type rit_type)
{
  zval *object = getThis();
  spl_recursive_it_object *intern;
  zval *iterator;
  zend_class_entry *ce_iterator;
  zend_long mode, flags;
  zend_error_handling error_handling;
  zval caching_it, aggregate_retval;

  zend_replace_error_handling(EH_THROW, spl_ce_InvalidArgumentException, &error_handling);

  switch (rit_type) {
    case RIT_RecursiveTreeIterator: {
      zval caching_it_flags, *user_caching_it_flags = NULL;
      mode = RIT_SELF_FIRST;
      flags = RTIT_BYPASS_KEY;

      if (zend_parse_parameters_ex(ZEND_PARSE_PARAMS_QUIET, ZEND_NUM_ARGS(), "o|lzl", &iterator, &flags, &user_caching_it_flags, &mode) == SUCCESS) {
        if (instanceof_function(Z_OBJCE_P(iterator), zend_ce_aggregate)) {
          zend_call_method_with_0_params(iterator, Z_OBJCE_P(iterator), &Z_OBJCE_P(iterator)->iterator_funcs.zf_new_iterator, "getiterator", &aggregate_retval);
          iterator = &aggregate_retval;
        } else {
          Z_ADDREF_P(iterator);
        }

        if (user_caching_it_flags) {
          ZVAL_COPY(&caching_it_flags, user_caching_it_flags);
        } else {
          ZVAL_LONG(&caching_it_flags, CIT_CATCH_GET_CHILD);
        }
        spl_instantiate_arg_ex2(spl_ce_RecursiveCachingIterator, &caching_it, iterator, &caching_it_flags);
        zval_ptr_dtor(&caching_it_flags);

        zval_ptr_dtor(iterator);
        iterator = &caching_it;
      } else {
        iterator = NULL;
      }
      break;
    }
    case RIT_RecursiveIteratorIterator:
    default: {
      mode = RIT_LEAVES_ONLY;
      flags = 0;

      if (zend_parse_parameters_ex(ZEND_PARSE_PARAMS_QUIET, ZEND_NUM_ARGS(), "o|ll", &iterator, &mode, &flags) == SUCCESS) {
        if (instanceof_function(Z_OBJCE_P(iterator), zend_ce_aggregate)) {
          zend_call_method_with_0_params(iterator, Z_OBJCE_P(iterator), &Z_OBJCE_P(iterator)->iterator_funcs.zf_new_iterator, "getiterator", &aggregate_retval);
          iterator = &aggregate_retval;
        } else {
          Z_ADDREF_P(iterator);
        }
      } else {
        iterator = NULL;
      }
      break;
    }
  }
  if (!iterator || !instanceof_function(Z_OBJCE_P(iterator), spl_ce_RecursiveIterator)) {
    if (iterator) {
      zval_ptr_dtor(iterator);
    }
    zend_throw_exception(spl_ce_InvalidArgumentException, "An instance of RecursiveIterator or IteratorAggregate creating it is required", 0);
    zend_restore_error_handling(&error_handling);
    return;
  }

  intern = Z_SPLRECURSIVE_IT_P(object);
  intern->iterators = emalloc(sizeof(spl_sub_iterator));
  intern->level = 0;
  intern->mode = mode;
  intern->flags = (int)flags;
  intern->max_depth = -1;
  intern->in_iteration = 0;
  intern->ce = Z_OBJCE_P(object);

  intern->beginIteration = zend_hash_str_find_ptr(&intern->ce->function_table, "beginiteration", sizeof("beginiteration") - 1);
  if (intern->beginIteration->common.scope == ce_base) {
    intern->beginIteration = NULL;
  }
  intern->endIteration = zend_hash_str_find_ptr(&intern->ce->function_table, "enditeration", sizeof("enditeration") - 1);
  if (intern->endIteration->common.scope == ce_base) {
    intern->endIteration = NULL;
  }
  intern->callHasChildren = zend_hash_str_find_ptr(&intern->ce->function_table, "callhaschildren", sizeof("callHasChildren") - 1);
  if (intern->callHasChildren->common.scope == ce_base) {
    intern->callHasChildren = NULL;
  }
  intern->callGetChildren = zend_hash_str_find_ptr(&intern->ce->function_table, "callgetchildren", sizeof("callGetChildren") - 1);
  if (intern->callGetChildren->common.scope == ce_base) {
    intern->callGetChildren = NULL;
  }
  intern->beginChildren = zend_hash_str_find_ptr(&intern->ce->function_table, "beginchildren", sizeof("beginchildren") - 1);
  if (intern->beginChildren->common.scope == ce_base) {
    intern->beginChildren = NULL;
  }
  intern->endChildren = zend_hash_str_find_ptr(&intern->ce->function_table, "endchildren", sizeof("endchildren") - 1);
  if (intern->endChildren->common.scope == ce_base) {
    intern->endChildren = NULL;
  }
  intern->nextElement = zend_hash_str_find_ptr(&intern->ce->function_table, "nextelement", sizeof("nextElement") - 1);
  if (intern->nextElement->common.scope == ce_base) {
    intern->nextElement = NULL;
  }

  ce_iterator = Z_OBJCE_P(iterator); /* respect inheritance, don't use spl_ce_RecursiveIterator */
  intern->iterators[0].iterator = ce_iterator->get_iterator(ce_iterator, iterator, 0);
  ZVAL_COPY_VALUE(&intern->iterators[0].zobject, iterator);
  intern->iterators[0].ce = ce_iterator;
  intern->iterators[0].state = RS_START;

  zend_restore_error_handling(&error_handling);

  if (EG(exception)) {
    zend_object_iterator *sub_iter;

    while (intern->level >= 0) {
      sub_iter = intern->iterators[intern->level].iterator;
      zend_iterator_dtor(sub_iter);
      zval_ptr_dtor(&intern->iterators[intern->level--].zobject);
    }
    efree(intern->iterators);
    intern->iterators = NULL;
  }
}

/* {{{ proto void RecursiveIteratorIterator::__construct(RecursiveIterator|IteratorAggregate it [, int mode = RIT_LEAVES_ONLY [, int flags = 0]]) throws InvalidArgumentException
   Creates a RecursiveIteratorIterator from a RecursiveIterator. */
SPL_METHOD(RecursiveIteratorIterator, __construct)
{
  spl_recursive_it_it_construct(INTERNAL_FUNCTION_PARAM_PASSTHRU, spl_ce_RecursiveIteratorIterator, zend_ce_iterator, RIT_RecursiveIteratorIterator);
} /* }}} */

/* {{{ proto void RecursiveIteratorIterator::rewind()
   Rewind the iterator to the first element of the top level inner iterator. */
SPL_METHOD(RecursiveIteratorIterator, rewind)
{
  spl_recursive_it_object   *object = Z_SPLRECURSIVE_IT_P(getThis());

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  spl_recursive_it_rewind_ex(object, getThis());
} /* }}} */

/* {{{ proto bool RecursiveIteratorIterator::valid()
   Check whether the current position is valid */
SPL_METHOD(RecursiveIteratorIterator, valid)
{
  spl_recursive_it_object   *object = Z_SPLRECURSIVE_IT_P(getThis());

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  RETURN_BOOL(spl_recursive_it_valid_ex(object, getThis()) == SUCCESS);
} /* }}} */

/* {{{ proto mixed RecursiveIteratorIterator::key()
   Access the current key */
SPL_METHOD(RecursiveIteratorIterator, key)
{
  spl_recursive_it_object   *object = Z_SPLRECURSIVE_IT_P(getThis());
  zend_object_iterator      *iterator;

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  SPL_FETCH_SUB_ITERATOR(iterator, object);

  if (iterator->funcs->get_current_key) {
    iterator->funcs->get_current_key(iterator, return_value);
  } else {
    RETURN_NULL();
  }
} /* }}} */

/* {{{ proto mixed RecursiveIteratorIterator::current()
   Access the current element value */
SPL_METHOD(RecursiveIteratorIterator, current)
{
  spl_recursive_it_object   *object = Z_SPLRECURSIVE_IT_P(getThis());
  zend_object_iterator      *iterator;
  zval                      *data;

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  SPL_FETCH_SUB_ITERATOR(iterator, object);

  data = iterator->funcs->get_current_data(iterator);
  if (data) {
    ZVAL_DEREF(data);
    ZVAL_COPY(return_value, data);
  }
} /* }}} */

/* {{{ proto void RecursiveIteratorIterator::next()
   Move forward to the next element */
SPL_METHOD(RecursiveIteratorIterator, next)
{
  spl_recursive_it_object   *object = Z_SPLRECURSIVE_IT_P(getThis());

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  spl_recursive_it_move_forward_ex(object, getThis());
} /* }}} */

/* {{{ proto int RecursiveIteratorIterator::getDepth()
   Get the current depth of the recursive iteration */
SPL_METHOD(RecursiveIteratorIterator, getDepth)
{
  spl_recursive_it_object   *object = Z_SPLRECURSIVE_IT_P(getThis());

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  RETURN_LONG(object->level);
} /* }}} */

/* {{{ proto RecursiveIterator RecursiveIteratorIterator::getSubIterator([int level])
   The current active sub iterator or the iterator at specified level */
SPL_METHOD(RecursiveIteratorIterator, getSubIterator)
{
  spl_recursive_it_object   *object = Z_SPLRECURSIVE_IT_P(getThis());
  zend_long  level = object->level;
  zval *value;

  if (zend_parse_parameters(ZEND_NUM_ARGS(), "|l", &level) == FAILURE) {
    return;
  }
  if (level < 0 || level > object->level) {
    RETURN_NULL();
  }

  if(!object->iterators) {
    zend_throw_exception_ex(spl_ce_LogicException, 0,
      "The object is in an invalid state as the parent constructor was not called");
    return;
  }

  value = &object->iterators[level].zobject;
  ZVAL_DEREF(value);
  ZVAL_COPY(return_value, value);
} /* }}} */

/* {{{ proto RecursiveIterator RecursiveIteratorIterator::getInnerIterator()
   The current active sub iterator */
SPL_METHOD(RecursiveIteratorIterator, getInnerIterator)
{
  spl_recursive_it_object   *object = Z_SPLRECURSIVE_IT_P(getThis());
  zval      *zobject;

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  SPL_FETCH_SUB_ELEMENT_ADDR(zobject, object, zobject);

  ZVAL_DEREF(zobject);
  ZVAL_COPY(return_value, zobject);
} /* }}} */
```

```c

/* {{{ proto RecursiveIterator RecursiveIteratorIterator::beginIteration()
   Called when iteration begins (after first rewind() call) */
SPL_METHOD(RecursiveIteratorIterator, beginIteration)
{
  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }
  /* nothing to do */
} /* }}} */

/* {{{ proto RecursiveIterator RecursiveIteratorIterator::endIteration()
   Called when iteration ends (when valid() first returns false */
SPL_METHOD(RecursiveIteratorIterator, endIteration)
{
  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }
  /* nothing to do */
} /* }}} */

/* {{{ proto bool RecursiveIteratorIterator::callHasChildren()
   Called for each element to test whether it has children */
SPL_METHOD(RecursiveIteratorIterator, callHasChildren)
{
  spl_recursive_it_object   *object = Z_SPLRECURSIVE_IT_P(getThis());
  zend_class_entry *ce;
  zval *zobject;

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  if (!object->iterators) {
    RETURN_NULL();
  }

  SPL_FETCH_SUB_ELEMENT(ce, object, ce);

  zobject = &object->iterators[object->level].zobject;
  if (Z_TYPE_P(zobject) == IS_UNDEF) {
    RETURN_FALSE;
  } else {
    zend_call_method_with_0_params(zobject, ce, NULL, "haschildren", return_value);
    if (Z_TYPE_P(return_value) == IS_UNDEF) {
      RETURN_FALSE;
    }
  }
} /* }}} */

/* {{{ proto RecursiveIterator RecursiveIteratorIterator::callGetChildren()
   Return children of current element */
SPL_METHOD(RecursiveIteratorIterator, callGetChildren)
{
  spl_recursive_it_object   *object = Z_SPLRECURSIVE_IT_P(getThis());
  zend_class_entry *ce;
  zval *zobject;

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  SPL_FETCH_SUB_ELEMENT(ce, object, ce);

  zobject = &object->iterators[object->level].zobject;
  if (Z_TYPE_P(zobject) == IS_UNDEF) {
    return;
  } else {
    zend_call_method_with_0_params(zobject, ce, NULL, "getchildren", return_value);
    if (Z_TYPE_P(return_value) == IS_UNDEF) {
      RETURN_NULL();
    }
  }
} /* }}} */

/* {{{ proto void RecursiveIteratorIterator::beginChildren()
   Called when recursing one level down */
SPL_METHOD(RecursiveIteratorIterator, beginChildren)
{
  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }
  /* nothing to do */
} /* }}} */

/* {{{ proto void RecursiveIteratorIterator::endChildren()
   Called when end recursing one level */
SPL_METHOD(RecursiveIteratorIterator, endChildren)
{
  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }
  /* nothing to do */
} /* }}} */

/* {{{ proto void RecursiveIteratorIterator::nextElement()
   Called when the next element is available */
SPL_METHOD(RecursiveIteratorIterator, nextElement)
{
  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }
  /* nothing to do */
} /* }}} */

/* {{{ proto void RecursiveIteratorIterator::setMaxDepth([$max_depth = -1])
   Set the maximum allowed depth (or any depth if pmax_depth = -1] */
SPL_METHOD(RecursiveIteratorIterator, setMaxDepth)
{
  spl_recursive_it_object   *object = Z_SPLRECURSIVE_IT_P(getThis());
  zend_long  max_depth = -1;

  if (zend_parse_parameters(ZEND_NUM_ARGS(), "|l", &max_depth) == FAILURE) {
    return;
  }
  if (max_depth < -1) {
    zend_throw_exception(spl_ce_OutOfRangeException, "Parameter max_depth must be >= -1", 0);
    return;
  } else if (max_depth > INT_MAX) {
    max_depth = INT_MAX;
  }

  object->max_depth = (int)max_depth;
} /* }}} */

/* {{{ proto int|false RecursiveIteratorIterator::getMaxDepth()
   Return the maximum accepted depth or false if any depth is allowed */
SPL_METHOD(RecursiveIteratorIterator, getMaxDepth)
{
  spl_recursive_it_object   *object = Z_SPLRECURSIVE_IT_P(getThis());

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  if (object->max_depth == -1) {
    RETURN_FALSE;
  } else {
    RETURN_LONG(object->max_depth);
  }
} /* }}} */

static union _zend_function *spl_recursive_it_get_method(zend_object **zobject, zend_string *method, const zval *key)
{
  union _zend_function    *function_handler;
  spl_recursive_it_object *object = spl_recursive_it_from_obj(*zobject);
  zend_long                     level = object->level;
  zval                    *zobj;

  if (!object->iterators) {
    php_error_docref(NULL, E_ERROR, "The %s instance wasn't initialized properly", ZSTR_VAL((*zobject)->ce->name));
  }
  zobj = &object->iterators[level].zobject;

  function_handler = std_object_handlers.get_method(zobject, method, key);
  if (!function_handler) {
    if ((function_handler = zend_hash_find_ptr(&Z_OBJCE_P(zobj)->function_table, method)) == NULL) {
      if (Z_OBJ_HT_P(zobj)->get_method) {
        *zobject = Z_OBJ_P(zobj);
        function_handler = (*zobject)->handlers->get_method(zobject, method, key);
      }
    } else {
      *zobject = Z_OBJ_P(zobj);
    }
  }
  return function_handler;
}

/* {{{ spl_RecursiveIteratorIterator_dtor */
static void spl_RecursiveIteratorIterator_dtor(zend_object *_object)
{
  spl_recursive_it_object *object = spl_recursive_it_from_obj(_object);
  zend_object_iterator *sub_iter;

  /* call standard dtor */
  zend_objects_destroy_object(_object);

  if (object->iterators) {
    while (object->level >= 0) {
      sub_iter = object->iterators[object->level].iterator;
      zend_iterator_dtor(sub_iter);
      zval_ptr_dtor(&object->iterators[object->level--].zobject);
    }
    efree(object->iterators);
    object->iterators = NULL;
  }
}
/* }}} */

/* {{{ spl_RecursiveIteratorIterator_free_storage */
static void spl_RecursiveIteratorIterator_free_storage(zend_object *_object)
{
  spl_recursive_it_object *object = spl_recursive_it_from_obj(_object);

  if (object->iterators) {
    efree(object->iterators);
    object->iterators = NULL;
    object->level     = 0;
  }

  zend_object_std_dtor(&object->std);
  smart_str_free(&object->prefix[0]);
  smart_str_free(&object->prefix[1]);
  smart_str_free(&object->prefix[2]);
  smart_str_free(&object->prefix[3]);
  smart_str_free(&object->prefix[4]);
  smart_str_free(&object->prefix[5]);

  smart_str_free(&object->postfix[0]);
}
/* }}} */

/* {{{ spl_RecursiveIteratorIterator_new_ex */
static zend_object *spl_RecursiveIteratorIterator_new_ex(zend_class_entry *class_type, int init_prefix)
{
  spl_recursive_it_object *intern;

  intern = zend_object_alloc(sizeof(spl_recursive_it_object), class_type);

  if (init_prefix) {
    smart_str_appendl(&intern->prefix[0], "",    0);
    smart_str_appendl(&intern->prefix[1], "| ",  2);
    smart_str_appendl(&intern->prefix[2], "  ",  2);
    smart_str_appendl(&intern->prefix[3], "|-",  2);
    smart_str_appendl(&intern->prefix[4], "\\-", 2);
    smart_str_appendl(&intern->prefix[5], "",    0);

    smart_str_appendl(&intern->postfix[0], "",    0);
  }

  zend_object_std_init(&intern->std, class_type);
  object_properties_init(&intern->std, class_type);

  intern->std.handlers = &spl_handlers_rec_it_it;
  return &intern->std;
}
/* }}} */

/* {{{ spl_RecursiveIteratorIterator_new */
static zend_object *spl_RecursiveIteratorIterator_new(zend_class_entry *class_type)
{
  return spl_RecursiveIteratorIterator_new_ex(class_type, 0);
}
/* }}} */

/* {{{ spl_RecursiveTreeIterator_new */
static zend_object *spl_RecursiveTreeIterator_new(zend_class_entry *class_type)
{
  return spl_RecursiveIteratorIterator_new_ex(class_type, 1);
}
/* }}} */

ZEND_BEGIN_ARG_INFO_EX(arginfo_recursive_it___construct, 0, 0, 1)
  ZEND_ARG_OBJ_INFO(0, iterator, Traversable, 0)
  ZEND_ARG_INFO(0, mode)
  ZEND_ARG_INFO(0, flags)
ZEND_END_ARG_INFO();

ZEND_BEGIN_ARG_INFO_EX(arginfo_recursive_it_getSubIterator, 0, 0, 0)
  ZEND_ARG_INFO(0, level)
ZEND_END_ARG_INFO();

ZEND_BEGIN_ARG_INFO_EX(arginfo_recursive_it_setMaxDepth, 0, 0, 0)
  ZEND_ARG_INFO(0, max_depth)
ZEND_END_ARG_INFO();

static const zend_function_entry spl_funcs_RecursiveIteratorIterator[] = {
  SPL_ME(RecursiveIteratorIterator, __construct,       arginfo_recursive_it___construct,    ZEND_ACC_PUBLIC)
  SPL_ME(RecursiveIteratorIterator, rewind,            arginfo_recursive_it_void,           ZEND_ACC_PUBLIC)
  SPL_ME(RecursiveIteratorIterator, valid,             arginfo_recursive_it_void,           ZEND_ACC_PUBLIC)
  SPL_ME(RecursiveIteratorIterator, key,               arginfo_recursive_it_void,           ZEND_ACC_PUBLIC)
  SPL_ME(RecursiveIteratorIterator, current,           arginfo_recursive_it_void,           ZEND_ACC_PUBLIC)
  SPL_ME(RecursiveIteratorIterator, next,              arginfo_recursive_it_void,           ZEND_ACC_PUBLIC)
  SPL_ME(RecursiveIteratorIterator, getDepth,          arginfo_recursive_it_void,           ZEND_ACC_PUBLIC)
  SPL_ME(RecursiveIteratorIterator, getSubIterator,    arginfo_recursive_it_getSubIterator, ZEND_ACC_PUBLIC)
  SPL_ME(RecursiveIteratorIterator, getInnerIterator,  arginfo_recursive_it_void,           ZEND_ACC_PUBLIC)
  SPL_ME(RecursiveIteratorIterator, beginIteration,    arginfo_recursive_it_void,           ZEND_ACC_PUBLIC)
  SPL_ME(RecursiveIteratorIterator, endIteration,      arginfo_recursive_it_void,           ZEND_ACC_PUBLIC)
  SPL_ME(RecursiveIteratorIterator, callHasChildren,   arginfo_recursive_it_void,           ZEND_ACC_PUBLIC)
  SPL_ME(RecursiveIteratorIterator, callGetChildren,   arginfo_recursive_it_void,           ZEND_ACC_PUBLIC)
  SPL_ME(RecursiveIteratorIterator, beginChildren,     arginfo_recursive_it_void,           ZEND_ACC_PUBLIC)
  SPL_ME(RecursiveIteratorIterator, endChildren,       arginfo_recursive_it_void,           ZEND_ACC_PUBLIC)
  SPL_ME(RecursiveIteratorIterator, nextElement,       arginfo_recursive_it_void,           ZEND_ACC_PUBLIC)
  SPL_ME(RecursiveIteratorIterator, setMaxDepth,       arginfo_recursive_it_setMaxDepth,    ZEND_ACC_PUBLIC)
  SPL_ME(RecursiveIteratorIterator, getMaxDepth,       arginfo_recursive_it_void,           ZEND_ACC_PUBLIC)
  PHP_FE_END
};

static void spl_recursive_tree_iterator_get_prefix(spl_recursive_it_object *object, zval *return_value)
{
  smart_str  str = {0};
  zval       has_next;
  int        level;

  smart_str_appendl(&str, ZSTR_VAL(object->prefix[0].s), ZSTR_LEN(object->prefix[0].s));

  for (level = 0; level < object->level; ++level) {
    zend_call_method_with_0_params(&object->iterators[level].zobject, object->iterators[level].ce, NULL, "hasnext", &has_next);
    if (Z_TYPE(has_next) != IS_UNDEF) {
      if (Z_TYPE(has_next) == IS_TRUE) {
        smart_str_appendl(&str, ZSTR_VAL(object->prefix[1].s), ZSTR_LEN(object->prefix[1].s));
      } else {
        smart_str_appendl(&str, ZSTR_VAL(object->prefix[2].s), ZSTR_LEN(object->prefix[2].s));
      }
      zval_ptr_dtor(&has_next);
    }
  }
  zend_call_method_with_0_params(&object->iterators[level].zobject, object->iterators[level].ce, NULL, "hasnext", &has_next);
  if (Z_TYPE(has_next) != IS_UNDEF) {
    if (Z_TYPE(has_next) == IS_TRUE) {
      smart_str_appendl(&str, ZSTR_VAL(object->prefix[3].s), ZSTR_LEN(object->prefix[3].s));
    } else {
      smart_str_appendl(&str, ZSTR_VAL(object->prefix[4].s), ZSTR_LEN(object->prefix[4].s));
    }
    zval_ptr_dtor(&has_next);
  }

  smart_str_appendl(&str, ZSTR_VAL(object->prefix[5].s), ZSTR_LEN(object->prefix[5].s));
  smart_str_0(&str);

  RETURN_NEW_STR(str.s);
}

static void spl_recursive_tree_iterator_get_entry(spl_recursive_it_object *object, zval *return_value)
{
  zend_object_iterator      *iterator = object->iterators[object->level].iterator;
  zval                      *data;
  zend_error_handling        error_handling;

  data = iterator->funcs->get_current_data(iterator);

  /* Replace exception handling so the catchable fatal error that is thrown when a class
   * without __toString is converted to string is converted into an exception. */
  zend_replace_error_handling(EH_THROW, spl_ce_UnexpectedValueException, &error_handling);
  if (data) {
    ZVAL_DEREF(data);
    if (Z_TYPE_P(data) == IS_ARRAY) {
      ZVAL_STRINGL(return_value, "Array", sizeof("Array")-1);
    } else {
      ZVAL_COPY(return_value, data);
      convert_to_string(return_value);
    }
  }
  zend_restore_error_handling(&error_handling);
}

static void spl_recursive_tree_iterator_get_postfix(spl_recursive_it_object *object, zval *return_value)
{
  RETVAL_STR(object->postfix[0].s);
  Z_ADDREF_P(return_value);
}

/* {{{ proto void RecursiveTreeIterator::__construct(RecursiveIterator|IteratorAggregate it [, int flags = RTIT_BYPASS_KEY [, int cit_flags = CIT_CATCH_GET_CHILD [, mode = RIT_SELF_FIRST ]]]) throws InvalidArgumentException
   RecursiveIteratorIterator to generate ASCII graphic trees for the entries in a RecursiveIterator */
SPL_METHOD(RecursiveTreeIterator, __construct)
{
  spl_recursive_it_it_construct(INTERNAL_FUNCTION_PARAM_PASSTHRU, spl_ce_RecursiveTreeIterator, zend_ce_iterator, RIT_RecursiveTreeIterator);
} /* }}} */

/* {{{ proto void RecursiveTreeIterator::setPrefixPart(int part, string prefix) throws OutOfRangeException
   Sets prefix parts as used in getPrefix() */
SPL_METHOD(RecursiveTreeIterator, setPrefixPart)
{
  zend_long  part;
  char* prefix;
  size_t   prefix_len;
  spl_recursive_it_object   *object = Z_SPLRECURSIVE_IT_P(getThis());

  if (zend_parse_parameters(ZEND_NUM_ARGS(), "ls", &part, &prefix, &prefix_len) == FAILURE) {
    return;
  }

  if (0 > part || part > 5) {
    zend_throw_exception_ex(spl_ce_OutOfRangeException, 0, "Use RecursiveTreeIterator::PREFIX_* constant");
    return;
  }

  smart_str_free(&object->prefix[part]);
  smart_str_appendl(&object->prefix[part], prefix, prefix_len);
} /* }}} */

/* {{{ proto string RecursiveTreeIterator::getPrefix()
   Returns the string to place in front of current element */
SPL_METHOD(RecursiveTreeIterator, getPrefix)
{
  spl_recursive_it_object   *object = Z_SPLRECURSIVE_IT_P(getThis());

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  if(!object->iterators) {
    zend_throw_exception_ex(spl_ce_LogicException, 0,
      "The object is in an invalid state as the parent constructor was not called");
    return;
  }

  spl_recursive_tree_iterator_get_prefix(object, return_value);
} /* }}} */

/* {{{ proto void RecursiveTreeIterator::setPostfix(string prefix)
   Sets postfix as used in getPostfix() */
SPL_METHOD(RecursiveTreeIterator, setPostfix)
{
  spl_recursive_it_object   *object = Z_SPLRECURSIVE_IT_P(getThis());
  char* postfix;
  size_t   postfix_len;

  if (zend_parse_parameters(ZEND_NUM_ARGS(), "s", &postfix, &postfix_len) == FAILURE) {
    return;
  }

  smart_str_free(&object->postfix[0]);
  smart_str_appendl(&object->postfix[0], postfix, postfix_len);
} /* }}} */

/* {{{ proto string RecursiveTreeIterator::getEntry()
   Returns the string presentation built for current element */
SPL_METHOD(RecursiveTreeIterator, getEntry)
{
  spl_recursive_it_object   *object = Z_SPLRECURSIVE_IT_P(getThis());

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  if(!object->iterators) {
    zend_throw_exception_ex(spl_ce_LogicException, 0,
      "The object is in an invalid state as the parent constructor was not called");
    return;
  }

  spl_recursive_tree_iterator_get_entry(object, return_value);
} /* }}} */

/* {{{ proto string RecursiveTreeIterator::getPostfix()
   Returns the string to place after the current element */
SPL_METHOD(RecursiveTreeIterator, getPostfix)
{
  spl_recursive_it_object   *object = Z_SPLRECURSIVE_IT_P(getThis());

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  if(!object->iterators) {
    zend_throw_exception_ex(spl_ce_LogicException, 0,
      "The object is in an invalid state as the parent constructor was not called");
    return;
  }

  spl_recursive_tree_iterator_get_postfix(object, return_value);
} /* }}} */

/* {{{ proto mixed RecursiveTreeIterator::current()
   Returns the current element prefixed and postfixed */
SPL_METHOD(RecursiveTreeIterator, current)
{
  spl_recursive_it_object   *object = Z_SPLRECURSIVE_IT_P(getThis());
  zval                       prefix, entry, postfix;
  char                      *ptr;
  zend_string               *str;

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  if(!object->iterators) {
    zend_throw_exception_ex(spl_ce_LogicException, 0,
      "The object is in an invalid state as the parent constructor was not called");
    return;
  }

  if (object->flags & RTIT_BYPASS_CURRENT) {
    zend_object_iterator      *iterator = object->iterators[object->level].iterator;
    zval                      *data;

        SPL_FETCH_SUB_ITERATOR(iterator, object);
    data = iterator->funcs->get_current_data(iterator);
    if (data) {
      ZVAL_DEREF(data);
      ZVAL_COPY(return_value, data);
      return;
    } else {
      RETURN_NULL();
    }
  }

  ZVAL_NULL(&prefix);
  ZVAL_NULL(&entry);
  spl_recursive_tree_iterator_get_prefix(object, &prefix);
  spl_recursive_tree_iterator_get_entry(object, &entry);
  if (Z_TYPE(entry) != IS_STRING) {
    zval_ptr_dtor(&prefix);
    zval_ptr_dtor(&entry);
    RETURN_NULL();
  }
  spl_recursive_tree_iterator_get_postfix(object, &postfix);

  str = zend_string_alloc(Z_STRLEN(prefix) + Z_STRLEN(entry) + Z_STRLEN(postfix), 0);
  ptr = ZSTR_VAL(str);

  memcpy(ptr, Z_STRVAL(prefix), Z_STRLEN(prefix));
  ptr += Z_STRLEN(prefix);
  memcpy(ptr, Z_STRVAL(entry), Z_STRLEN(entry));
  ptr += Z_STRLEN(entry);
  memcpy(ptr, Z_STRVAL(postfix), Z_STRLEN(postfix));
  ptr += Z_STRLEN(postfix);
  *ptr = 0;

  zval_ptr_dtor(&prefix);
  zval_ptr_dtor(&entry);
  zval_ptr_dtor(&postfix);

  RETURN_NEW_STR(str);
} /* }}} */

/* {{{ proto mixed RecursiveTreeIterator::key()
   Returns the current key prefixed and postfixed */
SPL_METHOD(RecursiveTreeIterator, key)
{
  spl_recursive_it_object   *object = Z_SPLRECURSIVE_IT_P(getThis());
  zend_object_iterator      *iterator;
  zval                       prefix, key, postfix, key_copy;
  char                      *ptr;
  zend_string               *str;

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  SPL_FETCH_SUB_ITERATOR(iterator, object);

  if (iterator->funcs->get_current_key) {
    iterator->funcs->get_current_key(iterator, &key);
  } else {
    ZVAL_NULL(&key);
  }

  if (object->flags & RTIT_BYPASS_KEY) {
    RETVAL_ZVAL(&key, 1, 1);
    return;
  }

  if (Z_TYPE(key) != IS_STRING) {
    if (zend_make_printable_zval(&key, &key_copy)) {
      key = key_copy;
    }
  }

  spl_recursive_tree_iterator_get_prefix(object, &prefix);
  spl_recursive_tree_iterator_get_postfix(object, &postfix);

  str = zend_string_alloc(Z_STRLEN(prefix) + Z_STRLEN(key) + Z_STRLEN(postfix), 0);
  ptr = ZSTR_VAL(str);

  memcpy(ptr, Z_STRVAL(prefix), Z_STRLEN(prefix));
  ptr += Z_STRLEN(prefix);
  memcpy(ptr, Z_STRVAL(key), Z_STRLEN(key));
  ptr += Z_STRLEN(key);
  memcpy(ptr, Z_STRVAL(postfix), Z_STRLEN(postfix));
  ptr += Z_STRLEN(postfix);
  *ptr = 0;

  zval_ptr_dtor(&prefix);
  zval_ptr_dtor(&key);
  zval_ptr_dtor(&postfix);

  RETURN_NEW_STR(str);
} /* }}} */

ZEND_BEGIN_ARG_INFO_EX(arginfo_recursive_tree_it___construct, 0, 0, 1)
  ZEND_ARG_OBJ_INFO(0, iterator, Traversable, 0)
  ZEND_ARG_INFO(0, flags)
  ZEND_ARG_INFO(0, caching_it_flags)
  ZEND_ARG_INFO(0, mode)
ZEND_END_ARG_INFO();

ZEND_BEGIN_ARG_INFO_EX(arginfo_recursive_tree_it_setPrefixPart, 0, 0, 2)
  ZEND_ARG_INFO(0, part)
  ZEND_ARG_INFO(0, value)
ZEND_END_ARG_INFO();

ZEND_BEGIN_ARG_INFO_EX(arginfo_recursive_tree_it_setPostfix, 0, 0, 1)
  ZEND_ARG_INFO(0, postfix)
ZEND_END_ARG_INFO();

static const zend_function_entry spl_funcs_RecursiveTreeIterator[] = {
  SPL_ME(RecursiveTreeIterator,     __construct,       arginfo_recursive_tree_it___construct,   ZEND_ACC_PUBLIC)
  SPL_ME(RecursiveIteratorIterator, rewind,            arginfo_recursive_it_void,               ZEND_ACC_PUBLIC)
  SPL_ME(RecursiveIteratorIterator, valid,             arginfo_recursive_it_void,               ZEND_ACC_PUBLIC)
  SPL_ME(RecursiveTreeIterator,     key,               arginfo_recursive_it_void,               ZEND_ACC_PUBLIC)
  SPL_ME(RecursiveTreeIterator,     current,           arginfo_recursive_it_void,               ZEND_ACC_PUBLIC)
  SPL_ME(RecursiveIteratorIterator, next,              arginfo_recursive_it_void,               ZEND_ACC_PUBLIC)
  SPL_ME(RecursiveIteratorIterator, beginIteration,    arginfo_recursive_it_void,               ZEND_ACC_PUBLIC)
  SPL_ME(RecursiveIteratorIterator, endIteration,      arginfo_recursive_it_void,               ZEND_ACC_PUBLIC)
  SPL_ME(RecursiveIteratorIterator, callHasChildren,   arginfo_recursive_it_void,               ZEND_ACC_PUBLIC)
  SPL_ME(RecursiveIteratorIterator, callGetChildren,   arginfo_recursive_it_void,               ZEND_ACC_PUBLIC)
  SPL_ME(RecursiveIteratorIterator, beginChildren,     arginfo_recursive_it_void,               ZEND_ACC_PUBLIC)
  SPL_ME(RecursiveIteratorIterator, endChildren,       arginfo_recursive_it_void,               ZEND_ACC_PUBLIC)
  SPL_ME(RecursiveIteratorIterator, nextElement,       arginfo_recursive_it_void,               ZEND_ACC_PUBLIC)
  SPL_ME(RecursiveTreeIterator,     getPrefix,         arginfo_recursive_it_void,               ZEND_ACC_PUBLIC)
  SPL_ME(RecursiveTreeIterator,     setPrefixPart,     arginfo_recursive_tree_it_setPrefixPart, ZEND_ACC_PUBLIC)
  SPL_ME(RecursiveTreeIterator,     getEntry,          arginfo_recursive_it_void,               ZEND_ACC_PUBLIC)
  SPL_ME(RecursiveTreeIterator,     setPostfix,        arginfo_recursive_tree_it_setPostfix,               ZEND_ACC_PUBLIC)
  SPL_ME(RecursiveTreeIterator,     getPostfix,        arginfo_recursive_it_void,               ZEND_ACC_PUBLIC)
  PHP_FE_END
};

#if MBO_0
static int spl_dual_it_gets_implemented(zend_class_entry *interface, zend_class_entry *class_type)
{
  class_type->iterator_funcs.zf_valid = NULL;
  class_type->iterator_funcs.zf_current = NULL;
  class_type->iterator_funcs.zf_key = NULL;
  class_type->iterator_funcs.zf_next = NULL;
  class_type->iterator_funcs.zf_rewind = NULL;
  if (!class_type->iterator_funcs.funcs) {
    class_type->iterator_funcs.funcs = &zend_interface_iterator_funcs_iterator;
  }

  return SUCCESS;
}
#endif

static union _zend_function *spl_dual_it_get_method(zend_object **object, zend_string *method, const zval *key)
{
  union _zend_function *function_handler;
  spl_dual_it_object   *intern;

  intern = spl_dual_it_from_obj(*object);

  function_handler = std_object_handlers.get_method(object, method, key);
  if (!function_handler && intern->inner.ce) {
    if ((function_handler = zend_hash_find_ptr(&intern->inner.ce->function_table, method)) == NULL) {
      if (Z_OBJ_HT(intern->inner.zobject)->get_method) {
        *object = Z_OBJ(intern->inner.zobject);
        function_handler = (*object)->handlers->get_method(object, method, key);
      }
    } else {
      *object = Z_OBJ(intern->inner.zobject);
    }
  }
  return function_handler;
}

#if MBO_0
int spl_dual_it_call_method(char *method, INTERNAL_FUNCTION_PARAMETERS)
{
  zval ***func_params, func;
  zval retval;
  int arg_count;
  int current = 0;
  int success;
  void **p;
  spl_dual_it_object   *intern;

  intern = Z_SPLDUAL_IT_P(getThis());

  ZVAL_STRING(&func, method, 0);

  p = EG(argument_stack).top_element-2;
  arg_count = (zend_ulong) *p;

  func_params = safe_emalloc(sizeof(zval **), arg_count, 0);

  current = 0;
  while (arg_count-- > 0) {
    func_params[current] = (zval **) p - (arg_count-current);
    current++;
  }
  arg_count = current; /* restore */

  if (call_user_function_ex(EG(function_table), NULL, &func, &retval, arg_count, func_params, 0, NULL) == SUCCESS && Z_TYPE(retval) != IS_UNDEF) {
    RETURN_ZVAL(&retval, 0, 0);

    success = SUCCESS;
  } else {
    zend_throw_error(NULL, "Unable to call %s::%s()", intern->inner.ce->name, method);
    success = FAILURE;
  }

  efree(func_params);
  return success;
}
#endif

#define SPL_CHECK_CTOR(intern, classname) \
  if (intern->dit_type == DIT_Unknown) { \
    zend_throw_exception_ex(spl_ce_BadMethodCallException, 0, "Classes derived from %s must call %s::__construct()", \
        ZSTR_VAL((spl_ce_##classname)->name), ZSTR_VAL((spl_ce_##classname)->name)); \
    return; \
  }

#define APPENDIT_CHECK_CTOR(intern) SPL_CHECK_CTOR(intern, AppendIterator)

static inline int spl_dual_it_fetch(spl_dual_it_object *intern, int check_more);

static inline int spl_cit_check_flags(zend_long flags)
{
  zend_long cnt = 0;

  cnt += (flags & CIT_CALL_TOSTRING) ? 1 : 0;
  cnt += (flags & CIT_TOSTRING_USE_KEY) ? 1 : 0;
  cnt += (flags & CIT_TOSTRING_USE_CURRENT) ? 1 : 0;
  cnt += (flags & CIT_TOSTRING_USE_INNER) ? 1 : 0;

  return cnt <= 1 ? SUCCESS : FAILURE;
}

static spl_dual_it_object* spl_dual_it_construct(INTERNAL_FUNCTION_PARAMETERS, zend_class_entry *ce_base, zend_class_entry *ce_inner, dual_it_type dit_
type)
{
  zval                 *zobject, retval;
  spl_dual_it_object   *intern;
  zend_class_entry     *ce = NULL;
  int                   inc_refcount = 1;
  zend_error_handling   error_handling;

  intern = Z_SPLDUAL_IT_P(getThis());

  if (intern->dit_type != DIT_Unknown) {
    zend_throw_exception_ex(spl_ce_BadMethodCallException, 0, "%s::getIterator() must be called exactly once per instance", ZSTR_VAL(ce_base->name));
    return NULL;
  }

  intern->dit_type = dit_type;
  switch (dit_type) {
    case DIT_LimitIterator: {
      intern->u.limit.offset = 0; /* start at beginning */
      intern->u.limit.count = -1; /* get all */
      if (zend_parse_parameters_throw(ZEND_NUM_ARGS(), "O|ll", &zobject, ce_inner, &intern->u.limit.offset, &intern->u.limit.count) == FAILURE) {
        return NULL;
      }
      if (intern->u.limit.offset < 0) {
        zend_throw_exception(spl_ce_OutOfRangeException, "Parameter offset must be >= 0", 0);
        return NULL;
      }
      if (intern->u.limit.count < 0 && intern->u.limit.count != -1) {
        zend_throw_exception(spl_ce_OutOfRangeException, "Parameter count must either be -1 or a value greater than or equal 0", 0);
        return NULL;
      }
      break;
    }
    case DIT_CachingIterator:
    case DIT_RecursiveCachingIterator: {
      zend_long flags = CIT_CALL_TOSTRING;
      if (zend_parse_parameters_throw(ZEND_NUM_ARGS(), "O|l", &zobject, ce_inner, &flags) == FAILURE) {
        return NULL;
      }
      if (spl_cit_check_flags(flags) != SUCCESS) {
        zend_throw_exception(spl_ce_InvalidArgumentException, "Flags must contain only one of CALL_TOSTRING, TOSTRING_USE_KEY, TOSTRING_USE_CURRENT, TOSTRING_USE_INNER", 0);
        return NULL;
      }
      intern->u.caching.flags |= flags & CIT_PUBLIC;
      array_init(&intern->u.caching.zcache);
      break;
    }
    case DIT_IteratorIterator: {
      zend_class_entry *ce_cast;
      zend_string *class_name;

      if (zend_parse_parameters_throw(ZEND_NUM_ARGS(), "O|S", &zobject, ce_inner, &class_name) == FAILURE) {
        return NULL;
      }
      ce = Z_OBJCE_P(zobject);
      if (!instanceof_function(ce, zend_ce_iterator)) {
        if (ZEND_NUM_ARGS() > 1) {
          if (!(ce_cast = zend_lookup_class(class_name))
          || !instanceof_function(ce, ce_cast)
          || !ce_cast->get_iterator
          ) {
            zend_throw_exception(spl_ce_LogicException, "Class to downcast to not found or not base class or does not implement Traversable", 0);
            return NULL;
          }
          ce = ce_cast;
        }
        if (instanceof_function(ce, zend_ce_aggregate)) {
          zend_call_method_with_0_params(zobject, ce, &ce->iterator_funcs.zf_new_iterator, "getiterator", &retval);
          if (EG(exception)) {
            zval_ptr_dtor(&retval);
            return NULL;
          }
          if (Z_TYPE(retval) != IS_OBJECT || !instanceof_function(Z_OBJCE(retval), zend_ce_traversable)) {
            zend_throw_exception_ex(spl_ce_LogicException, 0, "%s::getIterator() must return an object that implements Traversable", ZSTR_VAL(ce->name));
            return NULL;
          }
          zobject = &retval;
          ce = Z_OBJCE_P(zobject);
          inc_refcount = 0;
        }
      }
      break;
    }
    case DIT_AppendIterator:
      zend_replace_error_handling(EH_THROW, spl_ce_InvalidArgumentException, &error_handling);
      spl_instantiate(spl_ce_ArrayIterator, &intern->u.append.zarrayit);
      zend_call_method_with_0_params(&intern->u.append.zarrayit, spl_ce_ArrayIterator, &spl_ce_ArrayIterator->constructor, "__construct", NULL);
      intern->u.append.iterator = spl_ce_ArrayIterator->get_iterator(spl_ce_ArrayIterator, &intern->u.append.zarrayit, 0);
      zend_restore_error_handling(&error_handling);
      return intern;
#if HAVE_PCRE || HAVE_BUNDLED_PCRE
    case DIT_RegexIterator:
    case DIT_RecursiveRegexIterator: {
      zend_string *regex;
      zend_long mode = REGIT_MODE_MATCH;

      intern->u.regex.use_flags = ZEND_NUM_ARGS() >= 5;
      intern->u.regex.flags = 0;
      intern->u.regex.preg_flags = 0;
      if (zend_parse_parameters_throw(ZEND_NUM_ARGS(), "OS|lll", &zobject, ce_inner, &regex, &mode, &intern->u.regex.flags, &intern->u.regex.preg_flags) == FAILURE) {
        return NULL;
      }
      if (mode < 0 || mode >= REGIT_MODE_MAX) {
        zend_throw_exception_ex(spl_ce_InvalidArgumentException, 0, "Illegal mode " ZEND_LONG_FMT, mode);
        return NULL;
      }
      intern->u.regex.mode = mode;
      intern->u.regex.regex = zend_string_copy(regex);

      zend_replace_error_handling(EH_THROW, spl_ce_InvalidArgumentException, &error_handling);
      intern->u.regex.pce = pcre_get_compiled_regex_cache(regex);
      zend_restore_error_handling(&error_handling);

      if (intern->u.regex.pce == NULL) {
        /* pcre_get_compiled_regex_cache has already sent error */
        return NULL;
      }
      php_pcre_pce_incref(intern->u.regex.pce);
      break;
    }
#endif
    case DIT_CallbackFilterIterator:
    case DIT_RecursiveCallbackFilterIterator: {
      _spl_cbfilter_it_intern *cfi = emalloc(sizeof(*cfi));
      cfi->fci.object = NULL;
      if (zend_parse_parameters_throw(ZEND_NUM_ARGS(), "Of", &zobject, ce_inner, &cfi->fci, &cfi->fcc) == FAILURE) {
        efree(cfi);
        return NULL;
      }
      Z_TRY_ADDREF(cfi->fci.function_name);
      cfi->object = cfi->fcc.object;
      if (cfi->object) GC_ADDREF(cfi->object);
      intern->u.cbfilter = cfi;
      break;
    }
    default:
      if (zend_parse_parameters_throw(ZEND_NUM_ARGS(), "O", &zobject, ce_inner) == FAILURE) {
        return NULL;
      }
      break;
  }

  if (inc_refcount) {
    ZVAL_COPY(&intern->inner.zobject, zobject);
  } else {
    ZVAL_COPY_VALUE(&intern->inner.zobject, zobject);
  }

  intern->inner.ce = dit_type == DIT_IteratorIterator ? ce : Z_OBJCE_P(zobject);
  intern->inner.object = Z_OBJ_P(zobject);
  intern->inner.iterator = intern->inner.ce->get_iterator(intern->inner.ce, zobject, 0);

  return intern;
}

/* {{{ proto void FilterIterator::__construct(Iterator it)
   Create an Iterator from another iterator */
SPL_METHOD(FilterIterator, __construct)
{
  spl_dual_it_construct(INTERNAL_FUNCTION_PARAM_PASSTHRU, spl_ce_FilterIterator, zend_ce_iterator, DIT_FilterIterator);
} /* }}} */

/* {{{ proto void CallbackFilterIterator::__construct(Iterator it, callback func)
   Create an Iterator from another iterator */
SPL_METHOD(CallbackFilterIterator, __construct)
{
  spl_dual_it_construct(INTERNAL_FUNCTION_PARAM_PASSTHRU, spl_ce_CallbackFilterIterator, zend_ce_iterator, DIT_CallbackFilterIterator);
} /* }}} */

/* {{{ proto Iterator FilterIterator::getInnerIterator()
       proto Iterator CachingIterator::getInnerIterator()
       proto Iterator LimitIterator::getInnerIterator()
       proto Iterator ParentIterator::getInnerIterator()
   Get the inner iterator */
SPL_METHOD(dual_it, getInnerIterator)
{
  spl_dual_it_object   *intern;

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());

  if (!Z_ISUNDEF(intern->inner.zobject)) {
    zval *value = &intern->inner.zobject;

    ZVAL_DEREF(value);
    ZVAL_COPY(return_value, value);
  } else {
    RETURN_NULL();
  }
} /* }}} */

static inline void spl_dual_it_free(spl_dual_it_object *intern)
{
  if (intern->inner.iterator && intern->inner.iterator->funcs->invalidate_current) {
    intern->inner.iterator->funcs->invalidate_current(intern->inner.iterator);
  }
  if (Z_TYPE(intern->current.data) != IS_UNDEF) {
    zval_ptr_dtor(&intern->current.data);
    ZVAL_UNDEF(&intern->current.data);
  }
  if (Z_TYPE(intern->current.key) != IS_UNDEF) {
    zval_ptr_dtor(&intern->current.key);
    ZVAL_UNDEF(&intern->current.key);
  }
  if (intern->dit_type == DIT_CachingIterator || intern->dit_type == DIT_RecursiveCachingIterator) {
    if (Z_TYPE(intern->u.caching.zstr) != IS_UNDEF) {
      zval_ptr_dtor(&intern->u.caching.zstr);
      ZVAL_UNDEF(&intern->u.caching.zstr);
    }
    if (Z_TYPE(intern->u.caching.zchildren) != IS_UNDEF) {
      zval_ptr_dtor(&intern->u.caching.zchildren);
      ZVAL_UNDEF(&intern->u.caching.zchildren);
    }
  }
}

static inline void spl_dual_it_rewind(spl_dual_it_object *intern)
{
  spl_dual_it_free(intern);
  intern->current.pos = 0;
  if (intern->inner.iterator && intern->inner.iterator->funcs->rewind) {
    intern->inner.iterator->funcs->rewind(intern->inner.iterator);
  }
}

static inline int spl_dual_it_valid(spl_dual_it_object *intern)
{
  if (!intern->inner.iterator) {
    return FAILURE;
  }
  /* FAILURE / SUCCESS */
  return intern->inner.iterator->funcs->valid(intern->inner.iterator);
}

static inline int spl_dual_it_fetch(spl_dual_it_object *intern, int check_more)
{
  zval *data;

  spl_dual_it_free(intern);
  if (!check_more || spl_dual_it_valid(intern) == SUCCESS) {
    data = intern->inner.iterator->funcs->get_current_data(intern->inner.iterator);
    if (data) {
      ZVAL_COPY(&intern->current.data, data);
    }

    if (intern->inner.iterator->funcs->get_current_key) {
      intern->inner.iterator->funcs->get_current_key(intern->inner.iterator, &intern->current.key);
      if (EG(exception)) {
        zval_ptr_dtor(&intern->current.key);
        ZVAL_UNDEF(&intern->current.key);
      }
    } else {
      ZVAL_LONG(&intern->current.key, intern->current.pos);
    }
    return EG(exception) ? FAILURE : SUCCESS;
  }
  return FAILURE;
}

static inline void spl_dual_it_next(spl_dual_it_object *intern, int do_free)
{
  if (do_free) {
    spl_dual_it_free(intern);
  } else if (!intern->inner.iterator) {
    zend_throw_error(NULL, "The inner constructor wasn't initialized with an iterator instance");
    return;
  }
  intern->inner.iterator->funcs->move_forward(intern->inner.iterator);
  intern->current.pos++;
}

/* {{{ proto void ParentIterator::rewind()
       proto void IteratorIterator::rewind()
   Rewind the iterator
   */
SPL_METHOD(dual_it, rewind)
{
  spl_dual_it_object   *intern;

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());

  spl_dual_it_rewind(intern);
  spl_dual_it_fetch(intern, 1);
} /* }}} */

/* {{{ proto bool FilterIterator::valid()
       proto bool ParentIterator::valid()
       proto bool IteratorIterator::valid()
       proto bool NoRewindIterator::valid()
   Check whether the current element is valid */
SPL_METHOD(dual_it, valid)
{
  spl_dual_it_object   *intern;

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());

  RETURN_BOOL(Z_TYPE(intern->current.data) != IS_UNDEF);
} /* }}} */

/* {{{ proto mixed FilterIterator::key()
       proto mixed CachingIterator::key()
       proto mixed LimitIterator::key()
       proto mixed ParentIterator::key()
       proto mixed IteratorIterator::key()
       proto mixed NoRewindIterator::key()
       proto mixed AppendIterator::key()
   Get the current key */
SPL_METHOD(dual_it, key)
{
  spl_dual_it_object   *intern;

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());

  if (Z_TYPE(intern->current.key) != IS_UNDEF) {
    zval *value = &intern->current.key;

    ZVAL_DEREF(value);
    ZVAL_COPY(return_value, value);
  } else {
    RETURN_NULL();
  }
} /* }}} */

/* {{{ proto mixed FilterIterator::current()
       proto mixed CachingIterator::current()
       proto mixed LimitIterator::current()
       proto mixed ParentIterator::current()
       proto mixed IteratorIterator::current()
       proto mixed NoRewindIterator::current()
   Get the current element value */
SPL_METHOD(dual_it, current)
{
  spl_dual_it_object   *intern;

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());

  if (Z_TYPE(intern->current.data) != IS_UNDEF) {
    zval *value = &intern->current.data;

    ZVAL_DEREF(value);
    ZVAL_COPY(return_value, value);
  } else {
    RETURN_NULL();
  }
} /* }}} */

/* {{{ proto void ParentIterator::next()
       proto void IteratorIterator::next()
       proto void NoRewindIterator::next()
   Move the iterator forward */
SPL_METHOD(dual_it, next)
{
  spl_dual_it_object   *intern;

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());

  spl_dual_it_next(intern, 1);
  spl_dual_it_fetch(intern, 1);
} /* }}} */

static inline void spl_filter_it_fetch(zval *zthis, spl_dual_it_object *intern)
{
  zval retval;

  while (spl_dual_it_fetch(intern, 1) == SUCCESS) {
    zend_call_method_with_0_params(zthis, intern->std.ce, NULL, "accept", &retval);
    if (Z_TYPE(retval) != IS_UNDEF) {
      if (zend_is_true(&retval)) {
        zval_ptr_dtor(&retval);
        return;
      }
      zval_ptr_dtor(&retval);
    }
    if (EG(exception)) {
      return;
    }
    intern->inner.iterator->funcs->move_forward(intern->inner.iterator);
  }
  spl_dual_it_free(intern);
}

static inline void spl_filter_it_rewind(zval *zthis, spl_dual_it_object *intern)
{
  spl_dual_it_rewind(intern);
  spl_filter_it_fetch(zthis, intern);
}

static inline void spl_filter_it_next(zval *zthis, spl_dual_it_object *intern)
{
  spl_dual_it_next(intern, 1);
  spl_filter_it_fetch(zthis, intern);
}

/* {{{ proto void FilterIterator::rewind()
   Rewind the iterator */
SPL_METHOD(FilterIterator, rewind)
{
  spl_dual_it_object   *intern;

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
  spl_filter_it_rewind(getThis(), intern);
} /* }}} */

/* {{{ proto void FilterIterator::next()
   Move the iterator forward */
SPL_METHOD(FilterIterator, next)
{
  spl_dual_it_object   *intern;

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
  spl_filter_it_next(getThis(), intern);
} /* }}} */

/* {{{ proto void RecursiveCallbackFilterIterator::__construct(RecursiveIterator it, callback func)
   Create a RecursiveCallbackFilterIterator from a RecursiveIterator */
SPL_METHOD(RecursiveCallbackFilterIterator, __construct)
{
  spl_dual_it_construct(INTERNAL_FUNCTION_PARAM_PASSTHRU, spl_ce_RecursiveCallbackFilterIterator, spl_ce_RecursiveIterator, DIT_RecursiveCallbackFilterIterator);
} /* }}} */


/* {{{ proto void RecursiveFilterIterator::__construct(RecursiveIterator it)
   Create a RecursiveFilterIterator from a RecursiveIterator */
SPL_METHOD(RecursiveFilterIterator, __construct)
{
  spl_dual_it_construct(INTERNAL_FUNCTION_PARAM_PASSTHRU, spl_ce_RecursiveFilterIterator, spl_ce_RecursiveIterator, DIT_RecursiveFilterIterator);
} /* }}} */

/* {{{ proto bool RecursiveFilterIterator::hasChildren()
   Check whether the inner iterator's current element has children */
SPL_METHOD(RecursiveFilterIterator, hasChildren)
{
  spl_dual_it_object   *intern;
  zval                  retval;

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());

  zend_call_method_with_0_params(&intern->inner.zobject, intern->inner.ce, NULL, "haschildren", &retval);
  if (Z_TYPE(retval) != IS_UNDEF) {
    RETURN_ZVAL(&retval, 0, 1);
  } else {
    RETURN_FALSE;
  }
} /* }}} */

/* {{{ proto RecursiveFilterIterator RecursiveFilterIterator::getChildren()
   Return the inner iterator's children contained in a RecursiveFilterIterator */
SPL_METHOD(RecursiveFilterIterator, getChildren)
{
  spl_dual_it_object   *intern;
  zval                  retval;

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());

  zend_call_method_with_0_params(&intern->inner.zobject, intern->inner.ce, NULL, "getchildren", &retval);
  if (!EG(exception) && Z_TYPE(retval) != IS_UNDEF) {
    spl_instantiate_arg_ex1(Z_OBJCE_P(getThis()), return_value, &retval);
  }
  zval_ptr_dtor(&retval);
} /* }}} */

/* {{{ proto RecursiveCallbackFilterIterator RecursiveCallbackFilterIterator::getChildren()
   Return the inner iterator's children contained in a RecursiveCallbackFilterIterator */
SPL_METHOD(RecursiveCallbackFilterIterator, getChildren)
{
  spl_dual_it_object   *intern;
  zval                  retval;

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());

  zend_call_method_with_0_params(&intern->inner.zobject, intern->inner.ce, NULL, "getchildren", &retval);
  if (!EG(exception) && Z_TYPE(retval) != IS_UNDEF) {
    spl_instantiate_arg_ex2(Z_OBJCE_P(getThis()), return_value, &retval, &intern->u.cbfilter->fci.function_name);
  }
  zval_ptr_dtor(&retval);
} /* }}} */
/* {{{ proto void ParentIterator::__construct(RecursiveIterator it)
   Create a ParentIterator from a RecursiveIterator */
SPL_METHOD(ParentIterator, __construct)
{
  spl_dual_it_construct(INTERNAL_FUNCTION_PARAM_PASSTHRU, spl_ce_ParentIterator, spl_ce_RecursiveIterator, DIT_ParentIterator);
} /* }}} */

#if HAVE_PCRE || HAVE_BUNDLED_PCRE
/* {{{ proto void RegexIterator::__construct(Iterator it, string regex [, int mode [, int flags [, int preg_flags]]])
   Create an RegexIterator from another iterator and a regular expression */
SPL_METHOD(RegexIterator, __construct)
{
  spl_dual_it_construct(INTERNAL_FUNCTION_PARAM_PASSTHRU, spl_ce_RegexIterator, zend_ce_iterator, DIT_RegexIterator);
} /* }}} */

/* {{{ proto bool CallbackFilterIterator::accept()
   Calls the callback with the current value, the current key and the inner iterator as arguments */
SPL_METHOD(CallbackFilterIterator, accept)
{
  spl_dual_it_object     *intern = Z_SPLDUAL_IT_P(getThis());
  zend_fcall_info        *fci = &intern->u.cbfilter->fci;
  zend_fcall_info_cache  *fcc = &intern->u.cbfilter->fcc;
  zval                    params[3];

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  if (Z_TYPE(intern->current.data) == IS_UNDEF || Z_TYPE(intern->current.key) == IS_UNDEF) {
    RETURN_FALSE;
  }

  ZVAL_COPY_VALUE(&params[0], &intern->current.data);
  ZVAL_COPY_VALUE(&params[1], &intern->current.key);
  ZVAL_COPY_VALUE(&params[2], &intern->inner.zobject);

  fci->retval = return_value;
  fci->param_count = 3;
  fci->params = params;
  fci->no_separation = 0;

  if (zend_call_function(fci, fcc) != SUCCESS || Z_ISUNDEF_P(return_value)) {
    RETURN_FALSE;
  }

  if (EG(exception)) {
    RETURN_NULL();
  }

  /* zend_call_function may change args to IS_REF */
  ZVAL_COPY_VALUE(&intern->current.data, &params[0]);
  ZVAL_COPY_VALUE(&intern->current.key, &params[1]);
}
/* }}} */

/* {{{ proto bool RegexIterator::accept()
   Match (string)current() against regular expression */
SPL_METHOD(RegexIterator, accept)
{
  spl_dual_it_object *intern;
  zend_string *result, *subject;
  size_t count = 0;
  zval zcount, *replacement, tmp_replacement, rv;
  pcre2_match_data *match_data;
  pcre2_code *re;
  int rc;

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());

  if (Z_TYPE(intern->current.data) == IS_UNDEF) {
    RETURN_FALSE;
  }

  if (intern->u.regex.flags & REGIT_USE_KEY) {
    subject = zval_get_string(&intern->current.key);
  } else {
    if (Z_TYPE(intern->current.data) == IS_ARRAY) {
      RETURN_FALSE;
    }
    subject = zval_get_string(&intern->current.data);
  }

  switch (intern->u.regex.mode)
  {
    case REGIT_MODE_MAX: /* won't happen but makes compiler happy */
    case REGIT_MODE_MATCH:
      re = php_pcre_pce_re(intern->u.regex.pce);
      match_data = php_pcre_create_match_data(0, re);
      if (!match_data) {
        RETURN_FALSE;
      }
      rc = pcre2_match(re, (PCRE2_SPTR)ZSTR_VAL(subject), ZSTR_LEN(subject), 0, 0, match_data, php_pcre_mctx());
      RETVAL_BOOL(rc >= 0);
      php_pcre_free_match_data(match_data);
      break;

    case REGIT_MODE_ALL_MATCHES:
    case REGIT_MODE_GET_MATCH:
      zval_ptr_dtor(&intern->current.data);
      ZVAL_UNDEF(&intern->current.data);
      php_pcre_match_impl(intern->u.regex.pce, ZSTR_VAL(subject), ZSTR_LEN(subject), &zcount,
        &intern->current.data, intern->u.regex.mode == REGIT_MODE_ALL_MATCHES, intern->u.regex.use_flags, intern->u.regex.preg_flags, 0);
      RETVAL_BOOL(Z_LVAL(zcount) > 0);
      break;

    case REGIT_MODE_SPLIT:
      zval_ptr_dtor(&intern->current.data);
      ZVAL_UNDEF(&intern->current.data);
      php_pcre_split_impl(intern->u.regex.pce, subject, &intern->current.data, -1, intern->u.regex.preg_flags);
      count = zend_hash_num_elements(Z_ARRVAL(intern->current.data));
      RETVAL_BOOL(count > 1);
      break;

    case REGIT_MODE_REPLACE:
      replacement = zend_read_property(intern->std.ce, getThis(), "replacement", sizeof("replacement")-1, 1, &rv);
      if (Z_TYPE_P(replacement) != IS_STRING) {
        ZVAL_COPY(&tmp_replacement, replacement);
        convert_to_string(&tmp_replacement);
        replacement = &tmp_replacement;
      }
      result = php_pcre_replace_impl(intern->u.regex.pce, subject, ZSTR_VAL(subject), ZSTR_LEN(subject), Z_STR_P(replacement), -1, &count);

      if (intern->u.regex.flags & REGIT_USE_KEY) {
        zval_ptr_dtor(&intern->current.key);
        ZVAL_STR(&intern->current.key, result);
      } else {
        zval_ptr_dtor(&intern->current.data);
        ZVAL_STR(&intern->current.data, result);
      }

      if (replacement == &tmp_replacement) {
        zval_ptr_dtor(replacement);
      }
      RETVAL_BOOL(count > 0);
  }

  if (intern->u.regex.flags & REGIT_INVERTED) {
    RETVAL_BOOL(Z_TYPE_P(return_value) != IS_TRUE);
  }
  zend_string_release(subject);
} /* }}} */

/* {{{ proto string RegexIterator::getRegex()
   Returns current regular expression */
SPL_METHOD(RegexIterator, getRegex)
{
  spl_dual_it_object *intern = Z_SPLDUAL_IT_P(getThis());

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  RETURN_STR_COPY(intern->u.regex.regex);
} /* }}} */

/* {{{ proto bool RegexIterator::getMode()
   Returns current operation mode */
SPL_METHOD(RegexIterator, getMode)
{
  spl_dual_it_object *intern;

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());

  RETURN_LONG(intern->u.regex.mode);
} /* }}} */

/* {{{ proto bool RegexIterator::setMode(int new_mode)
   Set new operation mode */
SPL_METHOD(RegexIterator, setMode)
{
  spl_dual_it_object *intern;
  zend_long mode;

  if (zend_parse_parameters(ZEND_NUM_ARGS(), "l", &mode) == FAILURE) {
    return;
  }

  if (mode < 0 || mode >= REGIT_MODE_MAX) {
    zend_throw_exception_ex(spl_ce_InvalidArgumentException, 0, "Illegal mode " ZEND_LONG_FMT, mode);
    return;/* NULL */
  }

  SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());

  intern->u.regex.mode = mode;
} /* }}} */

/* {{{ proto bool RegexIterator::getFlags()
   Returns current operation flags */
SPL_METHOD(RegexIterator, getFlags)
{
  spl_dual_it_object *intern;

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());

  RETURN_LONG(intern->u.regex.flags);
} /* }}} */

/* {{{ proto bool RegexIterator::setFlags(int new_flags)
   Set operation flags */
SPL_METHOD(RegexIterator, setFlags)
{
  spl_dual_it_object *intern;
  zend_long flags;

  if (zend_parse_parameters(ZEND_NUM_ARGS(), "l", &flags) == FAILURE) {
    return;
  }

  SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());

  intern->u.regex.flags = flags;
} /* }}} */

/* {{{ proto bool RegexIterator::getFlags()
   Returns current PREG flags (if in use or NULL) */
SPL_METHOD(RegexIterator, getPregFlags)
{
  spl_dual_it_object *intern;

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());

  if (intern->u.regex.use_flags) {
    RETURN_LONG(intern->u.regex.preg_flags);
  } else {
    return;
  }
} /* }}} */

/* {{{ proto bool RegexIterator::setPregFlags(int new_flags)
   Set PREG flags */
SPL_METHOD(RegexIterator, setPregFlags)
{
  spl_dual_it_object *intern;
  zend_long preg_flags;

  if (zend_parse_parameters(ZEND_NUM_ARGS(), "l", &preg_flags) == FAILURE) {
    return;
  }

  SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());

  intern->u.regex.preg_flags = preg_flags;
  intern->u.regex.use_flags = 1;
} /* }}} */

/* {{{ proto void RecursiveRegexIterator::__construct(RecursiveIterator it, string regex [, int mode [, int flags [, int preg_flags]]])
   Create an RecursiveRegexIterator from another recursive iterator and a regular expression */
SPL_METHOD(RecursiveRegexIterator, __construct)
{
  spl_dual_it_construct(INTERNAL_FUNCTION_PARAM_PASSTHRU, spl_ce_RecursiveRegexIterator, spl_ce_RecursiveIterator, DIT_RecursiveRegexIterator);
} /* }}} */

```

```c
/* {{{ proto RecursiveRegexIterator RecursiveRegexIterator::getChildren()
   Return the inner iterator's children contained in a RecursiveRegexIterator */
SPL_METHOD(RecursiveRegexIterator, getChildren)
{
  spl_dual_it_object   *intern;
  zval                 retval;

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());

  zend_call_method_with_0_params(&intern->inner.zobject, intern->inner.ce, NULL, "getchildren", &retval);
  if (!EG(exception)) {
    zval args[5];

    ZVAL_COPY(&args[0], &retval);
    ZVAL_STR_COPY(&args[1], intern->u.regex.regex);
    ZVAL_LONG(&args[2], intern->u.regex.mode);
    ZVAL_LONG(&args[3], intern->u.regex.flags);
    ZVAL_LONG(&args[4], intern->u.regex.preg_flags);

    spl_instantiate_arg_n(Z_OBJCE_P(getThis()), return_value, 5, args);

    zval_ptr_dtor(&args[0]);
    zval_ptr_dtor(&args[1]);
  }
  zval_ptr_dtor(&retval);
} /* }}} */

SPL_METHOD(RecursiveRegexIterator, accept)
{
  spl_dual_it_object *intern;

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());

  if (Z_TYPE(intern->current.data) == IS_UNDEF) {
    RETURN_FALSE;
  } else if (Z_TYPE(intern->current.data) == IS_ARRAY) {
    RETURN_BOOL(zend_hash_num_elements(Z_ARRVAL(intern->current.data)) > 0);
  }

  zend_call_method_with_0_params(getThis(), spl_ce_RegexIterator, NULL, "accept", return_value);
}

#endif

/* {{{ spl_dual_it_dtor */
static void spl_dual_it_dtor(zend_object *_object)
{
  spl_dual_it_object *object = spl_dual_it_from_obj(_object);

  /* call standard dtor */
  zend_objects_destroy_object(_object);

  spl_dual_it_free(object);

  if (object->inner.iterator) {
    zend_iterator_dtor(object->inner.iterator);
  }
}
/* }}} */

/* {{{ spl_dual_it_free_storage */
static void spl_dual_it_free_storage(zend_object *_object)
{
  spl_dual_it_object *object = spl_dual_it_from_obj(_object);


  if (!Z_ISUNDEF(object->inner.zobject)) {
    zval_ptr_dtor(&object->inner.zobject);
  }

  if (object->dit_type == DIT_AppendIterator) {
    zend_iterator_dtor(object->u.append.iterator);
    if (Z_TYPE(object->u.append.zarrayit) != IS_UNDEF) {
      zval_ptr_dtor(&object->u.append.zarrayit);
    }
  }

  if (object->dit_type == DIT_CachingIterator || object->dit_type == DIT_RecursiveCachingIterator) {
    zval_ptr_dtor(&object->u.caching.zcache);
  }

#if HAVE_PCRE || HAVE_BUNDLED_PCRE
  if (object->dit_type == DIT_RegexIterator || object->dit_type == DIT_RecursiveRegexIterator) {
    if (object->u.regex.pce) {
      php_pcre_pce_decref(object->u.regex.pce);
    }
    if (object->u.regex.regex) {
      zend_string_release(object->u.regex.regex);
    }
  }
#endif

  if (object->dit_type == DIT_CallbackFilterIterator || object->dit_type == DIT_RecursiveCallbackFilterIterator) {
    if (object->u.cbfilter) {
      _spl_cbfilter_it_intern *cbfilter = object->u.cbfilter;
      object->u.cbfilter = NULL;
      zval_ptr_dtor(&cbfilter->fci.function_name);
      if (cbfilter->fci.object) {
        OBJ_RELEASE(cbfilter->fci.object);
      }
      efree(cbfilter);
    }
  }

  zend_object_std_dtor(&object->std);
}
/* }}} */

/* {{{ spl_dual_it_new */
static zend_object *spl_dual_it_new(zend_class_entry *class_type)
{
  spl_dual_it_object *intern;

  intern = zend_object_alloc(sizeof(spl_dual_it_object), class_type);
  intern->dit_type = DIT_Unknown;

  zend_object_std_init(&intern->std, class_type);
  object_properties_init(&intern->std, class_type);

  intern->std.handlers = &spl_handlers_dual_it;
  return &intern->std;
}
/* }}} */

ZEND_BEGIN_ARG_INFO(arginfo_filter_it___construct, 0)
  ZEND_ARG_OBJ_INFO(0, iterator, Iterator, 0)
ZEND_END_ARG_INFO();

static const zend_function_entry spl_funcs_FilterIterator[] = {
  SPL_ME(FilterIterator,  __construct,      arginfo_filter_it___construct, ZEND_ACC_PUBLIC)
  SPL_ME(FilterIterator,  rewind,           arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
  SPL_ME(dual_it,         valid,            arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
  SPL_ME(dual_it,         key,              arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
  SPL_ME(dual_it,         current,          arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
  SPL_ME(FilterIterator,  next,             arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
  SPL_ME(dual_it,         getInnerIterator, arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
  SPL_ABSTRACT_ME(FilterIterator, accept,   arginfo_recursive_it_void)
  PHP_FE_END
};

ZEND_BEGIN_ARG_INFO(arginfo_callback_filter_it___construct, 0)
  ZEND_ARG_OBJ_INFO(0, iterator, Iterator, 0)
  ZEND_ARG_INFO(0, callback)
ZEND_END_ARG_INFO();

static const zend_function_entry spl_funcs_CallbackFilterIterator[] = {
  SPL_ME(CallbackFilterIterator, __construct, arginfo_callback_filter_it___construct, ZEND_ACC_PUBLIC)
  SPL_ME(CallbackFilterIterator, accept,      arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
  PHP_FE_END
};

ZEND_BEGIN_ARG_INFO(arginfo_recursive_callback_filter_it___construct, 0)
  ZEND_ARG_OBJ_INFO(0, iterator, RecursiveIterator, 0)
  ZEND_ARG_INFO(0, callback)
ZEND_END_ARG_INFO();

static const zend_function_entry spl_funcs_RecursiveCallbackFilterIterator[] = {
  SPL_ME(RecursiveCallbackFilterIterator, __construct, arginfo_recursive_callback_filter_it___construct, ZEND_ACC_PUBLIC)
  SPL_ME(RecursiveFilterIterator,  hasChildren,      arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
  SPL_ME(RecursiveCallbackFilterIterator,  getChildren,      arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
  PHP_FE_END
};

ZEND_BEGIN_ARG_INFO(arginfo_parent_it___construct, 0)
  ZEND_ARG_OBJ_INFO(0, iterator, RecursiveIterator, 0)
ZEND_END_ARG_INFO();

static const zend_function_entry spl_funcs_RecursiveFilterIterator[] = {
  SPL_ME(RecursiveFilterIterator,  __construct,      arginfo_parent_it___construct, ZEND_ACC_PUBLIC)
  SPL_ME(RecursiveFilterIterator,  hasChildren,      arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
  SPL_ME(RecursiveFilterIterator,  getChildren,      arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
  PHP_FE_END
};

static const zend_function_entry spl_funcs_ParentIterator[] = {
  SPL_ME(ParentIterator,  __construct,      arginfo_parent_it___construct, ZEND_ACC_PUBLIC)
  SPL_MA(ParentIterator,  accept,           RecursiveFilterIterator, hasChildren, arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
  PHP_FE_END
};

#if HAVE_PCRE || HAVE_BUNDLED_PCRE
ZEND_BEGIN_ARG_INFO_EX(arginfo_regex_it___construct, 0, 0, 2)
  ZEND_ARG_OBJ_INFO(0, iterator, Iterator, 0)
  ZEND_ARG_INFO(0, regex)
  ZEND_ARG_INFO(0, mode)
  ZEND_ARG_INFO(0, flags)
  ZEND_ARG_INFO(0, preg_flags)
ZEND_END_ARG_INFO();

ZEND_BEGIN_ARG_INFO_EX(arginfo_regex_it_set_mode, 0, 0, 1)
  ZEND_ARG_INFO(0, mode)
ZEND_END_ARG_INFO();

ZEND_BEGIN_ARG_INFO_EX(arginfo_regex_it_set_flags, 0, 0, 1)
  ZEND_ARG_INFO(0, flags)
ZEND_END_ARG_INFO();

ZEND_BEGIN_ARG_INFO_EX(arginfo_regex_it_set_preg_flags, 0, 0, 1)
  ZEND_ARG_INFO(0, preg_flags)
ZEND_END_ARG_INFO();

static const zend_function_entry spl_funcs_RegexIterator[] = {
  SPL_ME(RegexIterator,   __construct,      arginfo_regex_it___construct,    ZEND_ACC_PUBLIC)
  SPL_ME(RegexIterator,   accept,           arginfo_recursive_it_void,       ZEND_ACC_PUBLIC)
  SPL_ME(RegexIterator,   getMode,          arginfo_recursive_it_void,       ZEND_ACC_PUBLIC)
  SPL_ME(RegexIterator,   setMode,          arginfo_regex_it_set_mode,       ZEND_ACC_PUBLIC)
  SPL_ME(RegexIterator,   getFlags,         arginfo_recursive_it_void,       ZEND_ACC_PUBLIC)
  SPL_ME(RegexIterator,   setFlags,         arginfo_regex_it_set_flags,      ZEND_ACC_PUBLIC)
  SPL_ME(RegexIterator,   getPregFlags,     arginfo_recursive_it_void,       ZEND_ACC_PUBLIC)
  SPL_ME(RegexIterator,   setPregFlags,     arginfo_regex_it_set_preg_flags, ZEND_ACC_PUBLIC)
  SPL_ME(RegexIterator,   getRegex,         arginfo_recursive_it_void,       ZEND_ACC_PUBLIC)
  PHP_FE_END
};

ZEND_BEGIN_ARG_INFO_EX(arginfo_rec_regex_it___construct, 0, 0, 2)
  ZEND_ARG_OBJ_INFO(0, iterator, RecursiveIterator, 0)
  ZEND_ARG_INFO(0, regex)
  ZEND_ARG_INFO(0, mode)
  ZEND_ARG_INFO(0, flags)
  ZEND_ARG_INFO(0, preg_flags)
ZEND_END_ARG_INFO();

static const zend_function_entry spl_funcs_RecursiveRegexIterator[] = {
  SPL_ME(RecursiveRegexIterator,  __construct,      arginfo_rec_regex_it___construct, ZEND_ACC_PUBLIC)
  SPL_ME(RecursiveRegexIterator,  accept,           arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
  SPL_ME(RecursiveFilterIterator, hasChildren,      arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
  SPL_ME(RecursiveRegexIterator,  getChildren,      arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
  PHP_FE_END
};
#endif

static inline int spl_limit_it_valid(spl_dual_it_object *intern)
{
  /* FAILURE / SUCCESS */
  if (intern->u.limit.count != -1 && intern->current.pos >= intern->u.limit.offset + intern->u.limit.count) {
    return FAILURE;
  } else {
    return spl_dual_it_valid(intern);
  }
}

static inline void spl_limit_it_seek(spl_dual_it_object *intern, zend_long pos)
{
  zval  zpos;

  spl_dual_it_free(intern);
  if (pos < intern->u.limit.offset) {
    zend_throw_exception_ex(spl_ce_OutOfBoundsException, 0, "Cannot seek to " ZEND_LONG_FMT " which is below the offset " ZEND_LONG_FMT, pos, intern->u.limit.offset);
    return;
  }
  if (pos >= intern->u.limit.offset + intern->u.limit.count && intern->u.limit.count != -1) {
    zend_throw_exception_ex(spl_ce_OutOfBoundsException, 0, "Cannot seek to " ZEND_LONG_FMT " which is behind offset " ZEND_LONG_FMT " plus count " ZEND_LONG_FMT, pos, intern->u.limit.offset, intern->u.limit.count);
    return;
  }
  if (pos != intern->current.pos && instanceof_function(intern->inner.ce, spl_ce_SeekableIterator)) {
    ZVAL_LONG(&zpos, pos);
    spl_dual_it_free(intern);
    zend_call_method_with_1_params(&intern->inner.zobject, intern->inner.ce, NULL, "seek", NULL, &zpos);
    zval_ptr_dtor(&zpos);
    if (!EG(exception)) {
      intern->current.pos = pos;
      if (spl_limit_it_valid(intern) == SUCCESS) {
        spl_dual_it_fetch(intern, 0);
      }
    }
  } else {
    /* emulate the forward seek, by next() calls */
    /* a back ward seek is done by a previous rewind() */
    if (pos < intern->current.pos) {
      spl_dual_it_rewind(intern);
    }
    while (pos > intern->current.pos && spl_dual_it_valid(intern) == SUCCESS) {
      spl_dual_it_next(intern, 1);
    }
    if (spl_dual_it_valid(intern) == SUCCESS) {
      spl_dual_it_fetch(intern, 1);
    }
  }
}

/* {{{ proto void LimitIterator::__construct(Iterator it [, int offset, int count])
   Construct a LimitIterator from an Iterator with a given starting offset and optionally a maximum count */
SPL_METHOD(LimitIterator, __construct)
{
  spl_dual_it_construct(INTERNAL_FUNCTION_PARAM_PASSTHRU, spl_ce_LimitIterator, zend_ce_iterator, DIT_LimitIterator);
} /* }}} */

/* {{{ proto void LimitIterator::rewind()
   Rewind the iterator to the specified starting offset */
SPL_METHOD(LimitIterator, rewind)
{
  spl_dual_it_object   *intern;

  SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
  spl_dual_it_rewind(intern);
  spl_limit_it_seek(intern, intern->u.limit.offset);
} /* }}} */

/* {{{ proto bool LimitIterator::valid()
   Check whether the current element is valid */
SPL_METHOD(LimitIterator, valid)
{
  spl_dual_it_object   *intern;

  SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());

/*  RETURN_BOOL(spl_limit_it_valid(intern) == SUCCESS);*/
  RETURN_BOOL((intern->u.limit.count == -1 || intern->current.pos < intern->u.limit.offset + intern->u.limit.count) && Z_TYPE(intern->current.data) != IS_UNDEF);
} /* }}} */

/* {{{ proto void LimitIterator::next()
   Move the iterator forward */
SPL_METHOD(LimitIterator, next)
{
  spl_dual_it_object   *intern;

  SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());

  spl_dual_it_next(intern, 1);
  if (intern->u.limit.count == -1 || intern->current.pos < intern->u.limit.offset + intern->u.limit.count) {
    spl_dual_it_fetch(intern, 1);
  }
} /* }}} */

/* {{{ proto void LimitIterator::seek(int position)
   Seek to the given position */
SPL_METHOD(LimitIterator, seek)
{
  spl_dual_it_object   *intern;
  zend_long                  pos;

  if (zend_parse_parameters(ZEND_NUM_ARGS(), "l", &pos) == FAILURE) {
    return;
  }

  SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
  spl_limit_it_seek(intern, pos);
  RETURN_LONG(intern->current.pos);
} /* }}} */

/* {{{ proto int LimitIterator::getPosition()
   Return the current position */
SPL_METHOD(LimitIterator, getPosition)
{
  spl_dual_it_object   *intern;
  SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
  RETURN_LONG(intern->current.pos);
} /* }}} */

ZEND_BEGIN_ARG_INFO(arginfo_seekable_it_seek, 0)
  ZEND_ARG_INFO(0, position)
ZEND_END_ARG_INFO();

static const zend_function_entry spl_funcs_SeekableIterator[] = {
  SPL_ABSTRACT_ME(SeekableIterator, seek, arginfo_seekable_it_seek)
  PHP_FE_END
};

ZEND_BEGIN_ARG_INFO_EX(arginfo_limit_it___construct, 0, 0, 1)
  ZEND_ARG_OBJ_INFO(0, iterator, Iterator, 0)
  ZEND_ARG_INFO(0, offset)
  ZEND_ARG_INFO(0, count)
ZEND_END_ARG_INFO();

ZEND_BEGIN_ARG_INFO(arginfo_limit_it_seek, 0)
```

```c
  ZEND_ARG_INFO(0, position)
ZEND_END_ARG_INFO();

static const zend_function_entry spl_funcs_LimitIterator[] = {
  SPL_ME(LimitIterator,   __construct,      arginfo_limit_it___construct, ZEND_ACC_PUBLIC)
  SPL_ME(LimitIterator,   rewind,           arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
  SPL_ME(LimitIterator,   valid,            arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
  SPL_ME(dual_it,         key,              arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
  SPL_ME(dual_it,         current,          arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
  SPL_ME(LimitIterator,   next,             arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
  SPL_ME(LimitIterator,   seek,             arginfo_limit_it_seek, ZEND_ACC_PUBLIC)
  SPL_ME(LimitIterator,   getPosition,      arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
  SPL_ME(dual_it,         getInnerIterator, arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
  PHP_FE_END
};

static inline int spl_caching_it_valid(spl_dual_it_object *intern)
{
  return intern->u.caching.flags & CIT_VALID ? SUCCESS : FAILURE;
}

static inline int spl_caching_it_has_next(spl_dual_it_object *intern)
{
  return spl_dual_it_valid(intern);
}

static inline void spl_caching_it_next(spl_dual_it_object *intern)
{
  if (spl_dual_it_fetch(intern, 1) == SUCCESS) {
    intern->u.caching.flags |= CIT_VALID;
    /* Full cache ? */
    if (intern->u.caching.flags & CIT_FULL_CACHE) {
      zval *key = &intern->current.key;
      zval *data = &intern->current.data;

      ZVAL_DEREF(data);
      Z_TRY_ADDREF_P(data);
      array_set_zval_key(Z_ARRVAL(intern->u.caching.zcache), key, data);
      zval_ptr_dtor(data);
    }
    /* Recursion ? */
    if (intern->dit_type == DIT_RecursiveCachingIterator) {
      zval retval, zchildren, zflags;
      zend_call_method_with_0_params(&intern->inner.zobject, intern->inner.ce, NULL, "haschildren", &retval);
      if (EG(exception)) {
        zval_ptr_dtor(&retval);
        if (intern->u.caching.flags & CIT_CATCH_GET_CHILD) {
          zend_clear_exception();
        } else {
          return;
        }
      } else {
        if (zend_is_true(&retval)) {
          zend_call_method_with_0_params(&intern->inner.zobject, intern->inner.ce, NULL, "getchildren", &zchildren);
          if (EG(exception)) {
            zval_ptr_dtor(&zchildren);
            if (intern->u.caching.flags & CIT_CATCH_GET_CHILD) {
              zend_clear_exception();
            } else {
              zval_ptr_dtor(&retval);
              return;
            }
          } else {
            ZVAL_LONG(&zflags, intern->u.caching.flags & CIT_PUBLIC);
            spl_instantiate_arg_ex2(spl_ce_RecursiveCachingIterator, &intern->u.caching.zchildren, &zchildren, &zflags);
            zval_ptr_dtor(&zchildren);
          }
        }
        zval_ptr_dtor(&retval);
        if (EG(exception)) {
          if (intern->u.caching.flags & CIT_CATCH_GET_CHILD) {
            zend_clear_exception();
          } else {
            return;
          }
        }
      }
    }
    if (intern->u.caching.flags & (CIT_TOSTRING_USE_INNER|CIT_CALL_TOSTRING)) {
      int  use_copy;
      zval expr_copy;
      if (intern->u.caching.flags & CIT_TOSTRING_USE_INNER) {
        ZVAL_COPY_VALUE(&intern->u.caching.zstr, &intern->inner.zobject);
      } else {
        ZVAL_COPY_VALUE(&intern->u.caching.zstr, &intern->current.data);
      }
      use_copy = zend_make_printable_zval(&intern->u.caching.zstr, &expr_copy);
      if (use_copy) {
        ZVAL_COPY_VALUE(&intern->u.caching.zstr, &expr_copy);
      } else {
        Z_TRY_ADDREF(intern->u.caching.zstr);
      }
    }
    spl_dual_it_next(intern, 0);
  } else {
    intern->u.caching.flags &= ~CIT_VALID;
  }
}

static inline void spl_caching_it_rewind(spl_dual_it_object *intern)
{
  spl_dual_it_rewind(intern);
  zend_hash_clean(Z_ARRVAL(intern->u.caching.zcache));
  spl_caching_it_next(intern);
}

/* {{{ proto void CachingIterator::__construct(Iterator it [, flags = CIT_CALL_TOSTRING])
   Construct a CachingIterator from an Iterator */
SPL_METHOD(CachingIterator, __construct)
{
  spl_dual_it_construct(INTERNAL_FUNCTION_PARAM_PASSTHRU, spl_ce_CachingIterator, zend_ce_iterator, DIT_CachingIterator);
} /* }}} */

/* {{{ proto void CachingIterator::rewind()
   Rewind the iterator */
SPL_METHOD(CachingIterator, rewind)
{
  spl_dual_it_object   *intern;

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());

  spl_caching_it_rewind(intern);
} /* }}} */

/* {{{ proto bool CachingIterator::valid()
   Check whether the current element is valid */
SPL_METHOD(CachingIterator, valid)
{
  spl_dual_it_object   *intern;

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());

  RETURN_BOOL(spl_caching_it_valid(intern) == SUCCESS);
} /* }}} */

/* {{{ proto void CachingIterator::next()
   Move the iterator forward */
SPL_METHOD(CachingIterator, next)
{
  spl_dual_it_object   *intern;

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());

  spl_caching_it_next(intern);
} /* }}} */

/* {{{ proto bool CachingIterator::hasNext()
   Check whether the inner iterator has a valid next element */
SPL_METHOD(CachingIterator, hasNext)
{
  spl_dual_it_object   *intern;

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());

  RETURN_BOOL(spl_caching_it_has_next(intern) == SUCCESS);
} /* }}} */

/* {{{ proto string CachingIterator::__toString()
   Return the string representation of the current element */
SPL_METHOD(CachingIterator, __toString)
{
  spl_dual_it_object   *intern;

  SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());

  if (!(intern->u.caching.flags & (CIT_CALL_TOSTRING|CIT_TOSTRING_USE_KEY|CIT_TOSTRING_USE_CURRENT|CIT_TOSTRING_USE_INNER)))  {
    zend_throw_exception_ex(spl_ce_BadMethodCallException, 0, "%s does not fetch string value (see CachingIterator::__construct)", ZSTR_VAL(Z_OBJCE_P(getThis())->name));
    return;
  }
  if (intern->u.caching.flags & CIT_TOSTRING_USE_KEY) {
    ZVAL_COPY(return_value, &intern->current.key);
    convert_to_string(return_value);
    return;
  } else if (intern->u.caching.flags & CIT_TOSTRING_USE_CURRENT) {
    ZVAL_COPY(return_value, &intern->current.data);
    convert_to_string(return_value);
    return;
  }
  if (Z_TYPE(intern->u.caching.zstr) == IS_STRING) {
    RETURN_STR_COPY(Z_STR_P(&intern->u.caching.zstr));
  } else {
    RETURN_EMPTY_STRING();
  }
} /* }}} */

/* {{{ proto void CachingIterator::offsetSet(mixed index, mixed newval)
   Set given index in cache */
SPL_METHOD(CachingIterator, offsetSet)
{
  spl_dual_it_object   *intern;
  zend_string *key;
  zval *value;

  SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());

  if (!(intern->u.caching.flags & CIT_FULL_CACHE))  {
    zend_throw_exception_ex(spl_ce_BadMethodCallException, 0, "%s does not use a full cache (see CachingIterator::__construct)", ZSTR_VAL(Z_OBJCE_P(getThis())->name));
    return;
  }

  if (zend_parse_parameters(ZEND_NUM_ARGS(), "Sz", &key, &value) == FAILURE) {
    return;
  }

  Z_TRY_ADDREF_P(value);
  zend_symtable_update(Z_ARRVAL(intern->u.caching.zcache), key, value);
}
/* }}} */

/* {{{ proto string CachingIterator::offsetGet(mixed index)
   Return the internal cache if used */
SPL_METHOD(CachingIterator, offsetGet)
{
  spl_dual_it_object   *intern;
  zend_string *key;
  zval *value;

  SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());

  if (!(intern->u.caching.flags & CIT_FULL_CACHE))  {
    zend_throw_exception_ex(spl_ce_BadMethodCallException, 0, "%s does not use a full cache (see CachingIterator::__construct)", ZSTR_VAL(Z_OBJCE_P(getThis())->name));
    return;
  }

  if (zend_parse_parameters(ZEND_NUM_ARGS(), "S", &key) == FAILURE) {
    return;
  }

  if ((value = zend_symtable_find(Z_ARRVAL(intern->u.caching.zcache), key)) == NULL) {
    zend_error(E_NOTICE, "Undefined index: %s", ZSTR_VAL(key));
    return;
  }

  ZVAL_DEREF(value);
  ZVAL_COPY(return_value, value);
}
/* }}} */

/* {{{ proto void CachingIterator::offsetUnset(mixed index)
   Unset given index in cache */
SPL_METHOD(CachingIterator, offsetUnset)
{
  spl_dual_it_object   *intern;
  zend_string *key;

  SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());

  if (!(intern->u.caching.flags & CIT_FULL_CACHE))  {
    zend_throw_exception_ex(spl_ce_BadMethodCallException, 0, "%s does not use a full cache (see CachingIterator::__construct)", ZSTR_VAL(Z_OBJCE_P(getThis())->name));
    return;
  }

  if (zend_parse_parameters(ZEND_NUM_ARGS(), "S", &key) == FAILURE) {
    return;
  }

  zend_symtable_del(Z_ARRVAL(intern->u.caching.zcache), key);
}
/* }}} */

/* {{{ proto bool CachingIterator::offsetExists(mixed index)
   Return whether the requested index exists */
SPL_METHOD(CachingIterator, offsetExists)
{
  spl_dual_it_object   *intern;
  zend_string *key;

  SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());

  if (!(intern->u.caching.flags & CIT_FULL_CACHE))  {
    zend_throw_exception_ex(spl_ce_BadMethodCallException, 0, "%s does not use a full cache (see CachingIterator::__construct)", ZSTR_VAL(Z_OBJCE_P(getThis())->name));
    return;
  }

  if (zend_parse_parameters(ZEND_NUM_ARGS(), "S", &key) == FAILURE) {
    return;
  }

  RETURN_BOOL(zend_symtable_exists(Z_ARRVAL(intern->u.caching.zcache), key));
}
/* }}} */

/* {{{ proto bool CachingIterator::getCache()
   Return the cache */
SPL_METHOD(CachingIterator, getCache)
{
  spl_dual_it_object   *intern;

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());

  if (!(intern->u.caching.flags & CIT_FULL_CACHE))  {
    zend_throw_exception_ex(spl_ce_BadMethodCallException, 0, "%s does not use a full cache (see CachingIterator::__construct)", ZSTR_VAL(Z_OBJCE_P(getThis())->name));
    return;
  }

  ZVAL_COPY(return_value, &intern->u.caching.zcache);
}
/* }}} */

/* {{{ proto int CachingIterator::getFlags()
   Return the internal flags */
SPL_METHOD(CachingIterator, getFlags)
{
  spl_dual_it_object   *intern;

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());

  RETURN_LONG(intern->u.caching.flags);
}
/* }}} */

/* {{{ proto void CachingIterator::setFlags(int flags)
   Set the internal flags */
SPL_METHOD(CachingIterator, setFlags)
{
  spl_dual_it_object   *intern;
  zend_long flags;

  SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());

  if (zend_parse_parameters(ZEND_NUM_ARGS(), "l", &flags) == FAILURE) {
    return;
  }

  if (spl_cit_check_flags(flags) != SUCCESS) {
    zend_throw_exception(spl_ce_InvalidArgumentException , "Flags must contain only one of CALL_TOSTRING, TOSTRING_USE_KEY, TOSTRING_USE_CURRENT, TOSTRING_USE_INNER", 0);
    return;
  }
  if ((intern->u.caching.flags & CIT_CALL_TOSTRING) != 0 && (flags & CIT_CALL_TOSTRING) == 0) {
    zend_throw_exception(spl_ce_InvalidArgumentException, "Unsetting flag CALL_TO_STRING is not possible", 0);
    return;
  }
  if ((intern->u.caching.flags & CIT_TOSTRING_USE_INNER) != 0 && (flags & CIT_TOSTRING_USE_INNER) == 0) {
    zend_throw_exception(spl_ce_InvalidArgumentException, "Unsetting flag TOSTRING_USE_INNER is not possible", 0);
    return;
  }
  if ((flags & CIT_FULL_CACHE) != 0 && (intern->u.caching.flags & CIT_FULL_CACHE) == 0) {
    /* clear on (re)enable */
    zend_hash_clean(Z_ARRVAL(intern->u.caching.zcache));
  }
  intern->u.caching.flags = (intern->u.caching.flags & ~CIT_PUBLIC) | (flags & CIT_PUBLIC);
}
/* }}} */

/* {{{ proto void CachingIterator::count()
```

```c
   Number of cached elements */
SPL_METHOD(CachingIterator, count)
{
  spl_dual_it_object   *intern;

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());

  if (!(intern->u.caching.flags & CIT_FULL_CACHE))  {
    zend_throw_exception_ex(spl_ce_BadMethodCallException, 0, "%s does not use a full cache (see CachingIterator::__construct)", ZSTR_VAL(Z_OBJCE_P(get
This())->name));
    return;
  }

  RETURN_LONG(zend_hash_num_elements(Z_ARRVAL(intern->u.caching.zcache)));
}
/* }}} */

ZEND_BEGIN_ARG_INFO_EX(arginfo_caching_it___construct, 0, 0, 1)
  ZEND_ARG_OBJ_INFO(0, iterator, Iterator, 0)
  ZEND_ARG_INFO(0, flags)
ZEND_END_ARG_INFO();

ZEND_BEGIN_ARG_INFO(arginfo_caching_it_setFlags, 0)
  ZEND_ARG_INFO(0, flags)
ZEND_END_ARG_INFO();

ZEND_BEGIN_ARG_INFO(arginfo_caching_it_offsetGet, 0)
  ZEND_ARG_INFO(0, index)
ZEND_END_ARG_INFO();

ZEND_BEGIN_ARG_INFO(arginfo_caching_it_offsetSet, 0)
  ZEND_ARG_INFO(0, index)
  ZEND_ARG_INFO(0, newval)
ZEND_END_ARG_INFO();

static const zend_function_entry spl_funcs_CachingIterator[] = {
  SPL_ME(CachingIterator, __construct,      arginfo_caching_it___construct, ZEND_ACC_PUBLIC)
  SPL_ME(CachingIterator, rewind,           arginfo_recursive_it_void,      ZEND_ACC_PUBLIC)
  SPL_ME(CachingIterator, valid,            arginfo_recursive_it_void,      ZEND_ACC_PUBLIC)
  SPL_ME(dual_it,         key,              arginfo_recursive_it_void,      ZEND_ACC_PUBLIC)
  SPL_ME(dual_it,         current,          arginfo_recursive_it_void,      ZEND_ACC_PUBLIC)
  SPL_ME(CachingIterator, next,             arginfo_recursive_it_void,      ZEND_ACC_PUBLIC)
  SPL_ME(CachingIterator, hasNext,          arginfo_recursive_it_void,      ZEND_ACC_PUBLIC)
  SPL_ME(CachingIterator, __toString,       arginfo_recursive_it_void,      ZEND_ACC_PUBLIC)
  SPL_ME(dual_it,         getInnerIterator, arginfo_recursive_it_void,      ZEND_ACC_PUBLIC)
  SPL_ME(CachingIterator, getFlags,         arginfo_recursive_it_void,      ZEND_ACC_PUBLIC)
  SPL_ME(CachingIterator, setFlags,         arginfo_caching_it_setFlags,    ZEND_ACC_PUBLIC)
  SPL_ME(CachingIterator, offsetGet,        arginfo_caching_it_offsetGet,   ZEND_ACC_PUBLIC)
  SPL_ME(CachingIterator, offsetSet,        arginfo_caching_it_offsetSet,   ZEND_ACC_PUBLIC)
  SPL_ME(CachingIterator, offsetUnset,      arginfo_caching_it_offsetGet,   ZEND_ACC_PUBLIC)
  SPL_ME(CachingIterator, offsetExists,     arginfo_caching_it_offsetGet,   ZEND_ACC_PUBLIC)
  SPL_ME(CachingIterator, getCache,         arginfo_recursive_it_void,      ZEND_ACC_PUBLIC)
  SPL_ME(CachingIterator, count,            arginfo_recursive_it_void,      ZEND_ACC_PUBLIC)
  PHP_FE_END
};

/* {{{ proto void RecursiveCachingIterator::__construct(RecursiveIterator it [, flags = CIT_CALL_TOSTRING])
   Create an iterator from a RecursiveIterator */
SPL_METHOD(RecursiveCachingIterator, __construct)
{
  spl_dual_it_construct(INTERNAL_FUNCTION_PARAM_PASSTHRU, spl_ce_RecursiveCachingIterator, spl_ce_RecursiveIterator, DIT_RecursiveCachingIterator);
} /* }}} */

/* {{{ proto bool RecursiveCachingIterator::hasChildren()
   Check whether the current element of the inner iterator has children */
SPL_METHOD(RecursiveCachingIterator, hasChildren)
{
  spl_dual_it_object   *intern;

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());

  RETURN_BOOL(Z_TYPE(intern->u.caching.zchildren) != IS_UNDEF);
} /* }}} */

/* {{{ proto RecursiveCachingIterator RecursiveCachingIterator::getChildren()
  Return the inner iterator's children as a RecursiveCachingIterator */
SPL_METHOD(RecursiveCachingIterator, getChildren)
{
  spl_dual_it_object   *intern;

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());

  if (Z_TYPE(intern->u.caching.zchildren) != IS_UNDEF) {
    zval *value = &intern->u.caching.zchildren;

    ZVAL_DEREF(value);
    ZVAL_COPY(return_value, value);
  } else {
    RETURN_NULL();
  }
} /* }}} */

ZEND_BEGIN_ARG_INFO_EX(arginfo_caching_rec_it___construct, 0, ZEND_RETURN_VALUE, 1)
  ZEND_ARG_OBJ_INFO(0, iterator, Iterator, 0)
  ZEND_ARG_INFO(0, flags)
ZEND_END_ARG_INFO();

static const zend_function_entry spl_funcs_RecursiveCachingIterator[] = {
  SPL_ME(RecursiveCachingIterator, __construct,   arginfo_caching_rec_it___construct, ZEND_ACC_PUBLIC)
  SPL_ME(RecursiveCachingIterator, hasChildren,   arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
  SPL_ME(RecursiveCachingIterator, getChildren,   arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
  PHP_FE_END
};

/* {{{ proto void IteratorIterator::__construct(Traversable it)
   Create an iterator from anything that is traversable */
SPL_METHOD(IteratorIterator, __construct)
{
  spl_dual_it_construct(INTERNAL_FUNCTION_PARAM_PASSTHRU, spl_ce_IteratorIterator, zend_ce_traversable, DIT_IteratorIterator);
} /* }}} */

ZEND_BEGIN_ARG_INFO(arginfo_iterator_it___construct, 0)
  ZEND_ARG_OBJ_INFO(0, iterator, Traversable, 0)
ZEND_END_ARG_INFO();

static const zend_function_entry spl_funcs_IteratorIterator[] = {
  SPL_ME(IteratorIterator, __construct,      arginfo_iterator_it___construct, ZEND_ACC_PUBLIC)
  SPL_ME(dual_it,          rewind,           arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
  SPL_ME(dual_it,          valid,            arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
  SPL_ME(dual_it,          key,              arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
  SPL_ME(dual_it,          current,          arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
  SPL_ME(dual_it,          next,             arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
  SPL_ME(dual_it,          getInnerIterator, arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
  PHP_FE_END
};

/* {{{ proto void NoRewindIterator::__construct(Iterator it)
   Create an iterator from another iterator */
SPL_METHOD(NoRewindIterator, __construct)
{
  spl_dual_it_construct(INTERNAL_FUNCTION_PARAM_PASSTHRU, spl_ce_NoRewindIterator, zend_ce_iterator, DIT_NoRewindIterator);
} /* }}} */

/* {{{ proto void NoRewindIterator::rewind()
   Prevent a call to inner iterators rewind() */
SPL_METHOD(NoRewindIterator, rewind)
{
  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }
  /* nothing to do */
} /* }}} */

/* {{{ proto bool NoRewindIterator::valid()
   Return inner iterators valid() */
SPL_METHOD(NoRewindIterator, valid)
{
  spl_dual_it_object   *intern;

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
  RETURN_BOOL(intern->inner.iterator->funcs->valid(intern->inner.iterator) == SUCCESS);
} /* }}} */

/* {{{ proto mixed NoRewindIterator::key()
   Return inner iterators key() */
SPL_METHOD(NoRewindIterator, key)
{
  spl_dual_it_object   *intern;

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());

  if (intern->inner.iterator->funcs->get_current_key) {
    intern->inner.iterator->funcs->get_current_key(intern->inner.iterator, return_value);
  } else {
    RETURN_NULL();
  }
} /* }}} */

/* {{{ proto mixed NoRewindIterator::current()
   Return inner iterators current() */
SPL_METHOD(NoRewindIterator, current)
{
  spl_dual_it_object   *intern;
  zval *data;

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
  data = intern->inner.iterator->funcs->get_current_data(intern->inner.iterator);
  if (data) {
    ZVAL_DEREF(data);
    ZVAL_COPY(return_value, data);
  }
} /* }}} */

/* {{{ proto void NoRewindIterator::next()
   Return inner iterators next() */
SPL_METHOD(NoRewindIterator, next)
{
  spl_dual_it_object   *intern;

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
  intern->inner.iterator->funcs->move_forward(intern->inner.iterator);
} /* }}} */

ZEND_BEGIN_ARG_INFO(arginfo_norewind_it___construct, 0)
  ZEND_ARG_OBJ_INFO(0, iterator, Iterator, 0)
ZEND_END_ARG_INFO();

static const zend_function_entry spl_funcs_NoRewindIterator[] = {
  SPL_ME(NoRewindIterator, __construct,      arginfo_norewind_it___construct, ZEND_ACC_PUBLIC)
  SPL_ME(NoRewindIterator, rewind,           arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
  SPL_ME(NoRewindIterator, valid,            arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
  SPL_ME(NoRewindIterator, key,              arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
  SPL_ME(NoRewindIterator, current,          arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
  SPL_ME(NoRewindIterator, next,             arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
  SPL_ME(dual_it,          getInnerIterator, arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
  PHP_FE_END
};

/* {{{ proto void InfiniteIterator::__construct(Iterator it)
   Create an iterator from another iterator */
SPL_METHOD(InfiniteIterator, __construct)
{
  spl_dual_it_construct(INTERNAL_FUNCTION_PARAM_PASSTHRU, spl_ce_InfiniteIterator, zend_ce_iterator, DIT_InfiniteIterator);
} /* }}} */

/* {{{ proto void InfiniteIterator::next()
   Prevent a call to inner iterators rewind() (internally the current data will be fetched if valid()) */
SPL_METHOD(InfiniteIterator, next)
{
  spl_dual_it_object   *intern;

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());

  spl_dual_it_next(intern, 1);
  if (spl_dual_it_valid(intern) == SUCCESS) {
    spl_dual_it_fetch(intern, 0);
  } else {
    spl_dual_it_rewind(intern);
    if (spl_dual_it_valid(intern) == SUCCESS) {
      spl_dual_it_fetch(intern, 0);
    }
  }
} /* }}} */

static const zend_function_entry spl_funcs_InfiniteIterator[] = {
  SPL_ME(InfiniteIterator, __construct,      arginfo_norewind_it___construct, ZEND_ACC_PUBLIC)
  SPL_ME(InfiniteIterator, next,             arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
  PHP_FE_END
};

/* {{{ proto void EmptyIterator::rewind()
   Does nothing  */
SPL_METHOD(EmptyIterator, rewind)
{
  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }
} /* }}} */

/* {{{ proto false EmptyIterator::valid()
   Return false */
SPL_METHOD(EmptyIterator, valid)
{
  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }
  RETURN_FALSE;
} /* }}} */

/* {{{ proto void EmptyIterator::key()
   Throws exception BadMethodCallException */
SPL_METHOD(EmptyIterator, key)
{
  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }
  zend_throw_exception(spl_ce_BadMethodCallException, "Accessing the key of an EmptyIterator", 0);
} /* }}} */

/* {{{ proto void EmptyIterator::current()
   Throws exception BadMethodCallException */
SPL_METHOD(EmptyIterator, current)
{
  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }
  zend_throw_exception(spl_ce_BadMethodCallException, "Accessing the value of an EmptyIterator", 0);
} /* }}} */

/* {{{ proto void EmptyIterator::next()
   Does nothing */
SPL_METHOD(EmptyIterator, next)
{
  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }
} /* }}} */

static const zend_function_entry spl_funcs_EmptyIterator[] = {
  SPL_ME(EmptyIterator, rewind,           arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
  SPL_ME(EmptyIterator, valid,            arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
  SPL_ME(EmptyIterator, key,              arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
  SPL_ME(EmptyIterator, current,          arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
  SPL_ME(EmptyIterator, next,             arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
  PHP_FE_END
};

int spl_append_it_next_iterator(spl_dual_it_object *intern) /* {{{*/
{
  spl_dual_it_free(intern);

  if (!Z_ISUNDEF(intern->inner.zobject)) {
    zval_ptr_dtor(&intern->inner.zobject);
    ZVAL_UNDEF(&intern->inner.zobject);
    intern->inner.ce = NULL;
    if (intern->inner.iterator) {
      zend_iterator_dtor(intern->inner.iterator);
      intern->inner.iterator = NULL;
    }
  }
  if (intern->u.append.iterator->funcs->valid(intern->u.append.iterator) == SUCCESS) {
    zval *it;

    it  = intern->u.append.iterator->funcs->get_current_data(intern->u.append.iterator);
    ZVAL_COPY(&intern->inner.zobject, it);
    intern->inner.ce = Z_OBJCE_P(it);
    intern->inner.iterator = intern->inner.ce->get_iterator(intern->inner.ce, it, 0);
    spl_dual_it_rewind(intern);
    return SUCCESS;
  } else {
    return FAILURE;
  }
} /* }}} */

void spl_append_it_fetch(spl_dual_it_object *intern) /* {{{*/
{
  while (spl_dual_it_valid(intern) != SUCCESS) {
    intern->u.append.iterator->funcs->move_forward(intern->u.append.iterator);
    if (spl_append_it_next_iterator(intern) != SUCCESS) {
      return;
    }
  }
  spl_dual_it_fetch(intern, 0);
} /* }}} */

void spl_append_it_next(spl_dual_it_object *intern) /* {{{ */
{
  if (spl_dual_it_valid(intern) == SUCCESS) {
    spl_dual_it_next(intern, 1);
  }
  spl_append_it_fetch(intern);
} /* }}} */

/* {{{ proto void AppendIterator::__construct()
```

```c
   Create an AppendIterator */
SPL_METHOD(AppendIterator, __construct)
{
  spl_dual_it_construct(INTERNAL_FUNCTION_PARAM_PASSTHRU, spl_ce_AppendIterator, zend_ce_iterator, DIT_AppendIterator);
} /* }}} */

/* {{{ proto void AppendIterator::append(Iterator it)
   Append an iterator */
SPL_METHOD(AppendIterator, append)
{
  spl_dual_it_object   *intern;
  zval *it;

  SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());

  if (zend_parse_parameters_ex(ZEND_PARSE_PARAMS_QUIET, ZEND_NUM_ARGS(), "O", &it, zend_ce_iterator) == FAILURE) {
    return;
  }
  if (intern->u.append.iterator->funcs->valid(intern->u.append.iterator) == SUCCESS && spl_dual_it_valid(intern) != SUCCESS) {
    spl_array_iterator_append(&intern->u.append.zarrayit, it);
    intern->u.append.iterator->funcs->move_forward(intern->u.append.iterator);
  }else{
    spl_array_iterator_append(&intern->u.append.zarrayit, it);
  }

  if (!intern->inner.iterator || spl_dual_it_valid(intern) != SUCCESS) {
    if (intern->u.append.iterator->funcs->valid(intern->u.append.iterator) != SUCCESS) {
      intern->u.append.iterator->funcs->rewind(intern->u.append.iterator);
    }
    do {
      spl_append_it_next_iterator(intern);
    } while (Z_OBJ(intern->inner.zobject) != Z_OBJ_P(it));
    spl_append_it_fetch(intern);
  }
} /* }}} */

/* {{{ proto mixed AppendIterator::current()
   Get the current element value */
SPL_METHOD(AppendIterator, current)
{
  spl_dual_it_object   *intern;

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());

  spl_dual_it_fetch(intern, 1);
  if (Z_TYPE(intern->current.data) != IS_UNDEF) {
    zval *value = &intern->current.data;

    ZVAL_DEREF(value);
    ZVAL_COPY(return_value, value);
  } else {
    RETURN_NULL();
  }
} /* }}} */

/* {{{ proto void AppendIterator::rewind()
   Rewind to the first iterator and rewind the first iterator, too */
SPL_METHOD(AppendIterator, rewind)
{
  spl_dual_it_object   *intern;

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());

  intern->u.append.iterator->funcs->rewind(intern->u.append.iterator);
  if (spl_append_it_next_iterator(intern) == SUCCESS) {
    spl_append_it_fetch(intern);
  }
} /* }}} */

/* {{{ proto bool AppendIterator::valid()
   Check if the current state is valid */
SPL_METHOD(AppendIterator, valid)
{
  spl_dual_it_object   *intern;

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());

  RETURN_BOOL(Z_TYPE(intern->current.data) != IS_UNDEF);
} /* }}} */

/* {{{ proto void AppendIterator::next()
   Forward to next element */
SPL_METHOD(AppendIterator, next)
{
  spl_dual_it_object   *intern;

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());

  spl_append_it_next(intern);
} /* }}} */

/* {{{ proto int AppendIterator::getIteratorIndex()
   Get index of iterator */
SPL_METHOD(AppendIterator, getIteratorIndex)
{
  spl_dual_it_object   *intern;

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());

  APPENDIT_CHECK_CTOR(intern);
  spl_array_iterator_key(&intern->u.append.zarrayit, return_value);
} /* }}} */

/* {{{ proto ArrayIterator AppendIterator::getArrayIterator()
   Get access to inner ArrayIterator */
SPL_METHOD(AppendIterator, getArrayIterator)
{
  spl_dual_it_object   *intern;
  zval *value;

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());

  value = &intern->u.append.zarrayit;
  ZVAL_DEREF(value);
  ZVAL_COPY(return_value, value);
} /* }}} */

ZEND_BEGIN_ARG_INFO(arginfo_append_it_append, 0)
  ZEND_ARG_OBJ_INFO(0, iterator, Iterator, 0)
ZEND_END_ARG_INFO();

static const zend_function_entry spl_funcs_AppendIterator[] = {
  SPL_ME(AppendIterator, __construct,      arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
  SPL_ME(AppendIterator, append,           arginfo_append_it_append, ZEND_ACC_PUBLIC)
  SPL_ME(AppendIterator, rewind,           arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
  SPL_ME(AppendIterator, valid,            arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
  SPL_ME(dual_it,        key,              arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
  SPL_ME(AppendIterator, current,          arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
  SPL_ME(AppendIterator, next,             arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
  SPL_ME(dual_it,        getInnerIterator, arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
  SPL_ME(AppendIterator, getIteratorIndex, arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
  SPL_ME(AppendIterator, getArrayIterator, arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
  PHP_FE_END
};

PHPAPI int spl_iterator_apply(zval *obj, spl_iterator_apply_func_t apply_func, void *puser)
{
  zend_object_iterator   *iter;
  zend_class_entry       *ce = Z_OBJCE_P(obj);

  iter = ce->get_iterator(ce, obj, 0);

  if (EG(exception)) {
    goto done;
  }

  iter->index = 0;
  if (iter->funcs->rewind) {
    iter->funcs->rewind(iter);
    if (EG(exception)) {
      goto done;
    }
  }

  while (iter->funcs->valid(iter) == SUCCESS) {
    if (EG(exception)) {
      goto done;
    }
    if (apply_func(iter, puser) == ZEND_HASH_APPLY_STOP || EG(exception)) {
      goto done;
    }
    iter->index++;
    iter->funcs->move_forward(iter);
    if (EG(exception)) {
      goto done;
    }
  }

done:
  if (iter) {
    zend_iterator_dtor(iter);
  }
  return EG(exception) ? FAILURE : SUCCESS;
}
/* }}} */

static int spl_iterator_to_array_apply(zend_object_iterator *iter, void *puser) /* {{{ */
{
  zval *data, *return_value = (zval*)puser;

  data = iter->funcs->get_current_data(iter);
  if (EG(exception)) {
    return ZEND_HASH_APPLY_STOP;
  }
  if (data == NULL) {
    return ZEND_HASH_APPLY_STOP;
  }
  if (iter->funcs->get_current_key) {
    zval key;
    iter->funcs->get_current_key(iter, &key);
    if (EG(exception)) {
      return ZEND_HASH_APPLY_STOP;
    }
    array_set_zval_key(Z_ARRVAL_P(return_value), &key, data);
    zval_ptr_dtor(&key);
  } else {
    Z_TRY_ADDREF_P(data);
    add_next_index_zval(return_value, data);
  }
  return ZEND_HASH_APPLY_KEEP;
}
/* }}} */

static int spl_iterator_to_values_apply(zend_object_iterator *iter, void *puser) /* {{{ */
{
  zval *data, *return_value = (zval*)puser;

  data = iter->funcs->get_current_data(iter);
  if (EG(exception)) {
    return ZEND_HASH_APPLY_STOP;
  }
  if (data == NULL) {
    return ZEND_HASH_APPLY_STOP;
  }
  Z_TRY_ADDREF_P(data);
  add_next_index_zval(return_value, data);
  return ZEND_HASH_APPLY_KEEP;
}
/* }}} */

/* {{{ proto array iterator_to_array(Traversable it [, bool use_keys = true])
   Copy the iterator into an array */
PHP_FUNCTION(iterator_to_array)
{
  zval  *obj;
  zend_bool use_keys = 1;

  if (zend_parse_parameters(ZEND_NUM_ARGS(), "O|b", &obj, zend_ce_traversable, &use_keys) == FAILURE) {
    RETURN_FALSE;
  }

  array_init(return_value);

  if (spl_iterator_apply(obj, use_keys ? spl_iterator_to_array_apply : spl_iterator_to_values_apply, (void*)return_value) != SUCCESS) {
    zval_ptr_dtor(return_value);
    RETURN_NULL();
  }
} /* }}} */

static int spl_iterator_count_apply(zend_object_iterator *iter, void *puser) /* {{{ */
{
  (*(zend_long*)puser)++;
  return ZEND_HASH_APPLY_KEEP;
}
/* }}} */

/* {{{ proto int iterator_count(Traversable it)
   Count the elements in an iterator */
PHP_FUNCTION(iterator_count)
{
  zval  *obj;
  zend_long  count = 0;

  if (zend_parse_parameters(ZEND_NUM_ARGS(), "O", &obj, zend_ce_traversable) == FAILURE) {
    RETURN_FALSE;
  }

  if (spl_iterator_apply(obj, spl_iterator_count_apply, (void*)&count) == SUCCESS) {
    RETURN_LONG(count);
  }
}
/* }}} */

typedef struct {
  zval                   *obj;
  zval                   *args;
  zend_long              count;
  zend_fcall_info        fci;
  zend_fcall_info_cache  fcc;
} spl_iterator_apply_info;

static int spl_iterator_func_apply(zend_object_iterator *iter, void *puser) /* {{{ */
{
  zval retval;
  spl_iterator_apply_info  *apply_info = (spl_iterator_apply_info*)puser;
  int result;

  apply_info->count++;
  zend_fcall_info_call(&apply_info->fci, &apply_info->fcc, &retval, NULL);
  if (Z_TYPE(retval) != IS_UNDEF) {
    result = zend_is_true(&retval) ? ZEND_HASH_APPLY_KEEP : ZEND_HASH_APPLY_STOP;
    zval_ptr_dtor(&retval);
  } else {
    result = ZEND_HASH_APPLY_STOP;
  }
  return result;
}
/* }}} */

/* {{{ proto int iterator_apply(Traversable it, mixed function [, mixed params])
   Calls a function for every element in an iterator */
PHP_FUNCTION(iterator_apply)
{
  spl_iterator_apply_info  apply_info;

  apply_info.args = NULL;
  if (zend_parse_parameters(ZEND_NUM_ARGS(), "Of|a!", &apply_info.obj, zend_ce_traversable, &apply_info.fci, &apply_info.fcc, &apply_info.args) == FAILURE) {
    return;
  }

  apply_info.count = 0;
  zend_fcall_info_args(&apply_info.fci, apply_info.args);
  if (spl_iterator_apply(apply_info.obj, spl_iterator_func_apply, (void*)&apply_info) == SUCCESS) {
    RETVAL_LONG(apply_info.count);
  } else {
    RETVAL_FALSE;
  }
  zend_fcall_info_args(&apply_info.fci, NULL);
}
/* }}} */

static const zend_function_entry spl_funcs_OuterIterator[] = {
  SPL_ABSTRACT_ME(OuterIterator, getInnerIterator,   arginfo_recursive_it_void)
  PHP_FE_END
};

/* {{{ PHP_MINIT_FUNCTION(spl_iterators)
 */
PHP_MINIT_FUNCTION(spl_iterators)
{
  REGISTER_SPL_INTERFACE(RecursiveIterator);
  REGISTER_SPL_ITERATOR(RecursiveIterator);

  REGISTER_SPL_STD_CLASS_EX(RecursiveIteratorIterator, spl_RecursiveIteratorIterator_new, spl_funcs_RecursiveIteratorIterator);
  REGISTER_SPL_ITERATOR(RecursiveIteratorIterator);

  memcpy(&spl_handlers_rec_it_it, zend_get_std_object_handlers(), sizeof(zend_object_handlers));
  spl_handlers_rec_it_it.offset = XtOffsetOf(spl_recursive_it_object, std);
  spl_handlers_rec_it_it.get_method = spl_recursive_it_get_method;
  spl_handlers_rec_it_it.clone_obj = NULL;
  spl_handlers_rec_it_it.dtor_obj = spl_RecursiveIteratorIterator_dtor;
  spl_handlers_rec_it_it.free_obj = spl_RecursiveIteratorIterator_free_storage;

  memcpy(&spl_handlers_dual_it, zend_get_std_object_handlers(), sizeof(zend_object_handlers));
  spl_handlers_dual_it.offset = XtOffsetOf(spl_dual_it_object, std);
  spl_handlers_dual_it.get_method = spl_dual_it_get_method;
  /*spl_handlers_dual_it.call_method = spl_dual_it_call_method;*/
  spl_handlers_dual_it.clone_obj = NULL;
  spl_handlers_dual_it.dtor_obj = spl_dual_it_dtor;
  spl_handlers_dual_it.free_obj = spl_dual_it_free_storage;

  spl_ce_RecursiveIteratorIterator->get_iterator = spl_recursive_it_get_iterator;
  spl_ce_RecursiveIteratorIterator->iterator_funcs.funcs = &spl_recursive_it_iterator_funcs;

  REGISTER_SPL_CLASS_CONST_LONG(RecursiveIteratorIterator, "LEAVES_ONLY",     RIT_LEAVES_ONLY);
  REGISTER_SPL_CLASS_CONST_LONG(RecursiveIteratorIterator, "SELF_FIRST",      RIT_SELF_FIRST);
  REGISTER_SPL_CLASS_CONST_LONG(RecursiveIteratorIterator, "CHILD_FIRST",     RIT_CHILD_FIRST);
  REGISTER_SPL_CLASS_CONST_LONG(RecursiveIteratorIterator, "CATCH_GET_CHILD", RIT_CATCH_GET_CHILD);

  REGISTER_SPL_INTERFACE(OuterIterator);
  REGISTER_SPL_ITERATOR(OuterIterator);

	REGISTER_SPL_STD_CLASS_EX(IteratorIterator, spl_dual_it_new, spl_funcs_IteratorIterator);
	REGISTER_SPL_ITERATOR(IteratorIterator);
	REGISTER_SPL_IMPLEMENTS(IteratorIterator, OuterIterator);

	REGISTER_SPL_SUB_CLASS_EX(FilterIterator, IteratorIterator, spl_dual_it_new, spl_funcs_FilterIterator);
	spl_ce_FilterIterator->ce_flags |= ZEND_ACC_EXPLICIT_ABSTRACT_CLASS;

	REGISTER_SPL_SUB_CLASS_EX(RecursiveFilterIterator, FilterIterator, spl_dual_it_new, spl_funcs_RecursiveFilterIterator);
	REGISTER_SPL_IMPLEMENTS(RecursiveFilterIterator, RecursiveIterator);

	REGISTER_SPL_SUB_CLASS_EX(CallbackFilterIterator, FilterIterator, spl_dual_it_new, spl_funcs_CallbackFilterIterator);

	REGISTER_SPL_SUB_CLASS_EX(RecursiveCallbackFilterIterator, CallbackFilterIterator, spl_dual_it_new, spl_funcs_RecursiveCallbackFilterIterator);
	REGISTER_SPL_IMPLEMENTS(RecursiveCallbackFilterIterator, RecursiveIterator);


	REGISTER_SPL_SUB_CLASS_EX(ParentIterator, RecursiveFilterIterator, spl_dual_it_new, spl_funcs_ParentIterator);

	REGISTER_SPL_INTERFACE(SeekableIterator);
	REGISTER_SPL_ITERATOR(SeekableIterator);

	REGISTER_SPL_SUB_CLASS_EX(LimitIterator, IteratorIterator, spl_dual_it_new, spl_funcs_LimitIterator);

	REGISTER_SPL_SUB_CLASS_EX(CachingIterator, IteratorIterator, spl_dual_it_new, spl_funcs_CachingIterator);
	REGISTER_SPL_IMPLEMENTS(CachingIterator, ArrayAccess);
	REGISTER_SPL_IMPLEMENTS(CachingIterator, Countable);

	REGISTER_SPL_CLASS_CONST_LONG(CachingIterator, "CALL_TOSTRING",        CIT_CALL_TOSTRING);
	REGISTER_SPL_CLASS_CONST_LONG(CachingIterator, "CATCH_GET_CHILD",      CIT_CATCH_GET_CHILD);
	REGISTER_SPL_CLASS_CONST_LONG(CachingIterator, "TOSTRING_USE_KEY",     CIT_TOSTRING_USE_KEY);
	REGISTER_SPL_CLASS_CONST_LONG(CachingIterator, "TOSTRING_USE_CURRENT", CIT_TOSTRING_USE_CURRENT);
	REGISTER_SPL_CLASS_CONST_LONG(CachingIterator, "TOSTRING_USE_INNER",   CIT_TOSTRING_USE_INNER);
	REGISTER_SPL_CLASS_CONST_LONG(CachingIterator, "FULL_CACHE",           CIT_FULL_CACHE);

	REGISTER_SPL_SUB_CLASS_EX(RecursiveCachingIterator, CachingIterator, spl_dual_it_new, spl_funcs_RecursiveCachingIterator);
	REGISTER_SPL_IMPLEMENTS(RecursiveCachingIterator, RecursiveIterator);

	REGISTER_SPL_SUB_CLASS_EX(NoRewindIterator, IteratorIterator, spl_dual_it_new, spl_funcs_NoRewindIterator);

	REGISTER_SPL_SUB_CLASS_EX(AppendIterator, IteratorIterator, spl_dual_it_new, spl_funcs_AppendIterator);

	REGISTER_SPL_IMPLEMENTS(RecursiveIteratorIterator, OuterIterator);

	REGISTER_SPL_SUB_CLASS_EX(InfiniteIterator, IteratorIterator, spl_dual_it_new, spl_funcs_InfiniteIterator);
#if HAVE_PCRE || HAVE_BUNDLED_PCRE
	REGISTER_SPL_SUB_CLASS_EX(RegexIterator, FilterIterator, spl_dual_it_new, spl_funcs_RegexIterator);
	REGISTER_SPL_CLASS_CONST_LONG(RegexIterator, "USE_KEY",     REGIT_USE_KEY);
	REGISTER_SPL_CLASS_CONST_LONG(RegexIterator, "INVERT_MATCH",REGIT_INVERTED);
	REGISTER_SPL_CLASS_CONST_LONG(RegexIterator, "MATCH",       REGIT_MODE_MATCH);
	REGISTER_SPL_CLASS_CONST_LONG(RegexIterator, "GET_MATCH",   REGIT_MODE_GET_MATCH);
	REGISTER_SPL_CLASS_CONST_LONG(RegexIterator, "ALL_MATCHES", REGIT_MODE_ALL_MATCHES);
	REGISTER_SPL_CLASS_CONST_LONG(RegexIterator, "SPLIT",       REGIT_MODE_SPLIT);
	REGISTER_SPL_CLASS_CONST_LONG(RegexIterator, "REPLACE",     REGIT_MODE_REPLACE);
	REGISTER_SPL_PROPERTY(RegexIterator, "replacement", 0);
	REGISTER_SPL_SUB_CLASS_EX(RecursiveRegexIterator, RegexIterator, spl_dual_it_new, spl_funcs_RecursiveRegexIterator);
	REGISTER_SPL_IMPLEMENTS(RecursiveRegexIterator, RecursiveIterator);
#else
	spl_ce_RegexIterator = NULL;
	spl_ce_RecursiveRegexIterator = NULL;
#endif

	REGISTER_SPL_STD_CLASS_EX(EmptyIterator, NULL, spl_funcs_EmptyIterator);
	REGISTER_SPL_ITERATOR(EmptyIterator);

	REGISTER_SPL_SUB_CLASS_EX(RecursiveTreeIterator, RecursiveIteratorIterator, spl_RecursiveTreeIterator_new, spl_funcs_RecursiveTreeIterator);
	REGISTER_SPL_CLASS_CONST_LONG(RecursiveTreeIterator, "BYPASS_CURRENT",      RTIT_BYPASS_CURRENT);
	REGISTER_SPL_CLASS_CONST_LONG(RecursiveTreeIterator, "BYPASS_KEY",          RTIT_BYPASS_KEY);
	REGISTER_SPL_CLASS_CONST_LONG(RecursiveTreeIterator, "PREFIX_LEFT",         0);
	REGISTER_SPL_CLASS_CONST_LONG(RecursiveTreeIterator, "PREFIX_MID_HAS_NEXT", 1);
	REGISTER_SPL_CLASS_CONST_LONG(RecursiveTreeIterator, "PREFIX_MID_LAST",     2);
	REGISTER_SPL_CLASS_CONST_LONG(RecursiveTreeIterator, "PREFIX_END_HAS_NEXT", 3);
	REGISTER_SPL_CLASS_CONST_LONG(RecursiveTreeIterator, "PREFIX_END_LAST",     4);
	REGISTER_SPL_CLASS_CONST_LONG(RecursiveTreeIterator, "PREFIX_RIGHT",        5);

	return SUCCESS;
}
/* }}} */

/*
 * Local variables:
 * tab-width: 4
 * c-basic-offset: 4
 * End:
 * vim600: fdm=marker
 * vim: noet sw=4 ts=4
 */
```

```c
/*
   +----------------------------------------------------------------------+
   | PHP Version 7                                                        |
   +----------------------------------------------------------------------+
   | Copyright (c) 1997-2018 The PHP Group                                |
   +----------------------------------------------------------------------+
   | This source file is subject to version 3.01 of the PHP license,      |
   | that is bundled with this package in the file LICENSE, and is        |
   | available through the world-wide-web at the following url:           |
   | http://www.php.net/license/3_01.txt                                  |
   | If you did not receive a copy of the PHP license and are unable to   |
   | obtain it through the world-wide-web, please send a note to          |
   | license@php.net so we can mail you a copy immediately.               |
   +----------------------------------------------------------------------+
   | Authors: Etienne Kneuss <colder@php.net>                             |
   +----------------------------------------------------------------------+
 */

/* $Id$ */

#ifndef SPL_DLLIST_H
#define SPL_DLLIST_H

#include "php.h"
#include "php_spl.h"

extern PHPAPI zend_class_entry *spl_ce_SplDoublyLinkedList;
extern PHPAPI zend_class_entry *spl_ce_SplQueue;
extern PHPAPI zend_class_entry *spl_ce_SplStack;

PHP_MINIT_FUNCTION(spl_dllist);

#endif /* SPL_DLLIST_H */

/*
 * Local Variables:
 * c-basic-offset: 4
 * tab-width: 4
 * End:
 * vim600: fdm=marker
 * vim: noet sw=4 ts=4
 */
```

```c
/*
   +----------------------------------------------------------------------+
   | PHP Version 7                                                        |
   +----------------------------------------------------------------------+
   | Copyright (c) 1997-2018 The PHP Group                                |
   +----------------------------------------------------------------------+
   | This source file is subject to version 3.01 of the PHP license,      |
   | that is bundled with this package in the file LICENSE, and is        |
   | available through the world-wide-web at the following url:           |
   | http://www.php.net/license/3_01.txt                                  |
   | If you did not receive a copy of the PHP license and are unable to   |
   | obtain it through the world-wide-web, please send a note to          |
   | license@php.net so we can mail you a copy immediately.               |
   +----------------------------------------------------------------------+
   | Authors: Marcus Boerger <helly@php.net>                              |
   +----------------------------------------------------------------------+
 */

/* $Id$ */

#ifndef PHP_FUNCTIONS_H
#define PHP_FUNCTIONS_H

#include "php.h"

typedef zend_object* (*create_object_func_t)(zend_class_entry *class_type);

#define REGISTER_SPL_STD_CLASS(class_name, obj_ctor) \
  spl_register_std_class(&spl_ce_ ## class_name, # class_name, obj_ctor, NULL);

#define REGISTER_SPL_STD_CLASS_EX(class_name, obj_ctor, funcs) \
  spl_register_std_class(&spl_ce_ ## class_name, # class_name, obj_ctor, funcs);

#define REGISTER_SPL_SUB_CLASS_EX(class_name, parent_class_name, obj_ctor, funcs) \
  spl_register_sub_class(&spl_ce_ ## class_name, spl_ce_ ## parent_class_name, # class_name, obj_ctor, funcs);

#define REGISTER_SPL_INTERFACE(class_name) \
  spl_register_interface(&spl_ce_ ## class_name, # class_name, spl_funcs_ ## class_name);

#define REGISTER_SPL_IMPLEMENTS(class_name, interface_name) \
  zend_class_implements(spl_ce_ ## class_name, 1, spl_ce_ ## interface_name);

#define REGISTER_SPL_ITERATOR(class_name) \
  zend_class_implements(spl_ce_ ## class_name, 1, zend_ce_iterator);

#define REGISTER_SPL_PROPERTY(class_name, prop_name, prop_flags) \
  spl_register_property(spl_ce_ ## class_name, prop_name, sizeof(prop_name)-1, prop_flags);

#define REGISTER_SPL_CLASS_CONST_LONG(class_name, const_name, value) \
  zend_declare_class_constant_long(spl_ce_ ## class_name, const_name, sizeof(const_name)-1, (zend_long)value);

void spl_register_std_class(zend_class_entry ** ppce, char * class_name, create_object_func_t ctor, const zend_function_entry * function_list);
void spl_register_sub_class(zend_class_entry ** ppce, zend_class_entry * parent_ce, char * class_name, create_object_func_t ctor, const zend_function_entry * function_list);
void spl_register_interface(zend_class_entry ** ppce, char * class_name, const zend_function_entry *functions);

void spl_register_property( zend_class_entry * class_entry, char *prop_name, int prop_name_len, int prop_flags);

/* sub: whether to allow subclasses/interfaces
   allow = 0: allow all classes and interfaces
   allow > 0: allow all that match and mask ce_flags
   allow < 0: disallow all that match and mask ce_flags
 */
void spl_add_class_name(zval * list, zend_class_entry * pce, int allow, int ce_flags);
void spl_add_interfaces(zval * list, zend_class_entry * pce, int allow, int ce_flags);
void spl_add_traits(zval * list, zend_class_entry * pce, int allow, int ce_flags);
int spl_add_classes(zend_class_entry *pce, zval *list, int sub, int allow, int ce_flags);

/* caller must efree(return) */
zend_string *spl_gen_private_prop_name(zend_class_entry *ce, char *prop_name, int prop_len);

#define SPL_ME(class_name, function_name, arg_info, flags) \
  PHP_ME( spl_ ## class_name, function_name, arg_info, flags)

#define SPL_ABSTRACT_ME(class_name, function_name, arg_info) \
  ZEND_ABSTRACT_ME( spl_ ## class_name, function_name, arg_info)

#define SPL_METHOD(class_name, function_name) \
  PHP_METHOD(spl_ ## class_name, function_name)

#define SPL_MA(class_name, function_name, alias_class, alias_function, arg_info, flags) \
  PHP_MALIAS(spl_ ## alias_class, function_name, alias_function, arg_info, flags)
#endif /* PHP_FUNCTIONS_H */

/*
 * Local Variables:
 * c-basic-offset: 4
 * tab-width: 4
 * End:
 * vim600: fdm=marker
 * vim: noet sw=4 ts=4
 */
```

```c
/*
   +----------------------------------------------------------------------+
   | PHP Version 7                                                        |
   +----------------------------------------------------------------------+
   | Copyright (c) 1997-2018 The PHP Group                                |
   +----------------------------------------------------------------------+
   | This source file is subject to version 3.01 of the PHP license,      |
   | that is bundled with this package in the file LICENSE, and is        |
   | available through the world-wide-web at the following url:          |
   | http://www.php.net/license/3_01.txt                                  |
   | If you did not receive a copy of the PHP license and are unable to   |
   | obtain it through the world-wide-web, please send a note to          |
   | license@php.net so we can mail you a copy immediately.               |
   +----------------------------------------------------------------------+
   | Authors: Marcus Boerger <helly@php.net>                              |
   +----------------------------------------------------------------------+
 */

/* $Id$ */

#ifdef HAVE_CONFIG_H
#include "config.h"
#endif

#include "php.h"
#include "php_ini.h"
#include "php_main.h"
#include "ext/standard/info.h"
#include "php_spl.h"
#include "spl_functions.h"
#include "spl_engine.h"
#include "spl_array.h"
#include "spl_directory.h"
#include "spl_iterators.h"
#include "spl_exceptions.h"
#include "spl_observer.h"
#include "spl_dllist.h"
#include "spl_fixedarray.h"
#include "spl_heap.h"
#include "zend_exceptions.h"
#include "zend_interfaces.h"
#include "ext/standard/php_mt_rand.h"
#include "main/snprintf.h"

#ifdef COMPILE_DL_SPL
ZEND_GET_MODULE(spl)
#endif

ZEND_DECLARE_MODULE_GLOBALS(spl)

#define SPL_DEFAULT_FILE_EXTENSIONS ".inc,.php"

/* {{{ PHP_GINIT_FUNCTION
 */
static PHP_GINIT_FUNCTION(spl)
{
  spl_globals->autoload_extensions     = NULL;
  spl_globals->autoload_functions      = NULL;
  spl_globals->autoload_running        = 0;
}
/* }}} */

static zend_class_entry * spl_find_ce_by_name(zend_string *name, zend_bool autoload)
{
  zend_class_entry *ce;

  if (!autoload) {
    zend_string *lc_name = zend_string_tolower(name);

    ce = zend_hash_find_ptr(EG(class_table), lc_name);
    zend_string_free(lc_name);
  } else {
    ce = zend_lookup_class(name);
  }
  if (ce == NULL) {
    php_error_docref(NULL, E_WARNING, "Class %s does not exist%s", ZSTR_VAL(name), autoload ? " and could not be loaded" : "");
    return NULL;
  }

  return ce;
}

/* {{{ proto array class_parents(object instance [, bool autoload = true])
 Return an array containing the names of all parent classes */
PHP_FUNCTION(class_parents)
{
  zval *obj;
  zend_class_entry *parent_class, *ce;
  zend_bool autoload = 1;

  if (zend_parse_parameters(ZEND_NUM_ARGS(), "z|b", &obj, &autoload) == FAILURE) {
    RETURN_FALSE;
  }

  if (Z_TYPE_P(obj) != IS_OBJECT && Z_TYPE_P(obj) != IS_STRING) {
    php_error_docref(NULL, E_WARNING, "object or string expected");
    RETURN_FALSE;
  }

  if (Z_TYPE_P(obj) == IS_STRING) {
    if (NULL == (ce = spl_find_ce_by_name(Z_STR_P(obj), autoload))) {
      RETURN_FALSE;
    }
  } else {
    ce = Z_OBJCE_P(obj);
  }

  array_init(return_value);
  parent_class = ce->parent;
  while (parent_class) {
    spl_add_class_name(return_value, parent_class, 0, 0);
    parent_class = parent_class->parent;
  }
}
/* }}} */

/* {{{ proto array class_implements(mixed what [, bool autoload ])
 Return all classes and interfaces implemented by SPL */
PHP_FUNCTION(class_implements)
{
  zval *obj;
  zend_bool autoload = 1;
  zend_class_entry *ce;

  if (zend_parse_parameters(ZEND_NUM_ARGS(), "z|b", &obj, &autoload) == FAILURE) {
    RETURN_FALSE;
  }
  if (Z_TYPE_P(obj) != IS_OBJECT && Z_TYPE_P(obj) != IS_STRING) {
    php_error_docref(NULL, E_WARNING, "object or string expected");
    RETURN_FALSE;
  }

  if (Z_TYPE_P(obj) == IS_STRING) {
    if (NULL == (ce = spl_find_ce_by_name(Z_STR_P(obj), autoload))) {
      RETURN_FALSE;
    }
  } else {
    ce = Z_OBJCE_P(obj);
  }

  array_init(return_value);
  spl_add_interfaces(return_value, ce, 1, ZEND_ACC_INTERFACE);
}
/* }}} */

/* {{{ proto array class_uses(mixed what [, bool autoload ])
 Return all traits used by a class. */
PHP_FUNCTION(class_uses)
{
  zval *obj;
  zend_bool autoload = 1;
  zend_class_entry *ce;

  if (zend_parse_parameters(ZEND_NUM_ARGS(), "z|b", &obj, &autoload) == FAILURE) {
    RETURN_FALSE;
  }
  if (Z_TYPE_P(obj) != IS_OBJECT && Z_TYPE_P(obj) != IS_STRING) {
    php_error_docref(NULL, E_WARNING, "object or string expected");
    RETURN_FALSE;
  }

  if (Z_TYPE_P(obj) == IS_STRING) {
    if (NULL == (ce = spl_find_ce_by_name(Z_STR_P(obj), autoload))) {
      RETURN_FALSE;
    }
  } else {
    ce = Z_OBJCE_P(obj);
  }

  array_init(return_value);
  spl_add_traits(return_value, ce, 1, ZEND_ACC_TRAIT);
}
/* }}} */

#define SPL_ADD_CLASS(class_name, z_list, sub, allow, ce_flags) \
  spl_add_classes(spl_ce_ ## class_name, z_list, sub, allow, ce_flags)

#define SPL_LIST_CLASSES(z_list, sub, allow, ce_flags) \
  SPL_ADD_CLASS(AppendIterator, z_list, sub, allow, ce_flags); \
  SPL_ADD_CLASS(ArrayIterator, z_list, sub, allow, ce_flags); \
  SPL_ADD_CLASS(ArrayObject, z_list, sub, allow, ce_flags); \
  SPL_ADD_CLASS(BadFunctionCallException, z_list, sub, allow, ce_flags); \
  SPL_ADD_CLASS(BadMethodCallException, z_list, sub, allow, ce_flags); \
  SPL_ADD_CLASS(CachingIterator, z_list, sub, allow, ce_flags); \
  SPL_ADD_CLASS(CallbackFilterIterator, z_list, sub, allow, ce_flags); \
  SPL_ADD_CLASS(DirectoryIterator, z_list, sub, allow, ce_flags); \
  SPL_ADD_CLASS(DomainException, z_list, sub, allow, ce_flags); \
  SPL_ADD_CLASS(EmptyIterator, z_list, sub, allow, ce_flags); \
  SPL_ADD_CLASS(FilesystemIterator, z_list, sub, allow, ce_flags); \
  SPL_ADD_CLASS(FilterIterator, z_list, sub, allow, ce_flags); \
  SPL_ADD_CLASS(GlobIterator, z_list, sub, allow, ce_flags); \
  SPL_ADD_CLASS(InfiniteIterator, z_list, sub, allow, ce_flags); \
  SPL_ADD_CLASS(InvalidArgumentException, z_list, sub, allow, ce_flags); \
  SPL_ADD_CLASS(IteratorIterator, z_list, sub, allow, ce_flags); \
  SPL_ADD_CLASS(LengthException, z_list, sub, allow, ce_flags); \
  SPL_ADD_CLASS(LimitIterator, z_list, sub, allow, ce_flags); \
  SPL_ADD_CLASS(LogicException, z_list, sub, allow, ce_flags); \
  SPL_ADD_CLASS(MultipleIterator, z_list, sub, allow, ce_flags); \
  SPL_ADD_CLASS(NoRewindIterator, z_list, sub, allow, ce_flags); \
  SPL_ADD_CLASS(OuterIterator, z_list, sub, allow, ce_flags); \
  SPL_ADD_CLASS(OutOfBoundsException, z_list, sub, allow, ce_flags); \
  SPL_ADD_CLASS(OutOfRangeException, z_list, sub, allow, ce_flags); \
  SPL_ADD_CLASS(OverflowException, z_list, sub, allow, ce_flags); \
  SPL_ADD_CLASS(ParentIterator, z_list, sub, allow, ce_flags); \
  SPL_ADD_CLASS(RangeException, z_list, sub, allow, ce_flags); \
  SPL_ADD_CLASS(RecursiveArrayIterator, z_list, sub, allow, ce_flags); \
  SPL_ADD_CLASS(RecursiveCachingIterator, z_list, sub, allow, ce_flags); \
  SPL_ADD_CLASS(RecursiveCallbackFilterIterator, z_list, sub, allow, ce_flags); \
  SPL_ADD_CLASS(RecursiveDirectoryIterator, z_list, sub, allow, ce_flags); \
  SPL_ADD_CLASS(RecursiveFilterIterator, z_list, sub, allow, ce_flags); \
  SPL_ADD_CLASS(RecursiveIterator, z_list, sub, allow, ce_flags); \
  SPL_ADD_CLASS(RecursiveIteratorIterator, z_list, sub, allow, ce_flags); \
  SPL_ADD_CLASS(RecursiveRegexIterator, z_list, sub, allow, ce_flags); \
  SPL_ADD_CLASS(RecursiveTreeIterator, z_list, sub, allow, ce_flags); \
  SPL_ADD_CLASS(RegexIterator, z_list, sub, allow, ce_flags); \
  SPL_ADD_CLASS(RuntimeException, z_list, sub, allow, ce_flags); \
  SPL_ADD_CLASS(SeekableIterator, z_list, sub, allow, ce_flags); \
  SPL_ADD_CLASS(SplDoublyLinkedList, z_list, sub, allow, ce_flags); \
  SPL_ADD_CLASS(SplFileInfo, z_list, sub, allow, ce_flags); \
  SPL_ADD_CLASS(SplFileObject, z_list, sub, allow, ce_flags); \
  SPL_ADD_CLASS(SplFixedArray, z_list, sub, allow, ce_flags); \
  SPL_ADD_CLASS(SplHeap, z_list, sub, allow, ce_flags); \
  SPL_ADD_CLASS(SplMinHeap, z_list, sub, allow, ce_flags); \
  SPL_ADD_CLASS(SplMaxHeap, z_list, sub, allow, ce_flags); \
  SPL_ADD_CLASS(SplObjectStorage, z_list, sub, allow, ce_flags); \
  SPL_ADD_CLASS(SplObserver, z_list, sub, allow, ce_flags); \
  SPL_ADD_CLASS(SplPriorityQueue, z_list, sub, allow, ce_flags); \
  SPL_ADD_CLASS(SplQueue, z_list, sub, allow, ce_flags); \
  SPL_ADD_CLASS(SplStack, z_list, sub, allow, ce_flags); \
  SPL_ADD_CLASS(SplSubject, z_list, sub, allow, ce_flags); \
  SPL_ADD_CLASS(SplTempFileObject, z_list, sub, allow, ce_flags); \
  SPL_ADD_CLASS(UnderflowException, z_list, sub, allow, ce_flags); \
  SPL_ADD_CLASS(UnexpectedValueException, z_list, sub, allow, ce_flags); \

/* {{{ proto array spl_classes()
 Return an array containing the names of all clsses and interfaces defined in SPL */
PHP_FUNCTION(spl_classes)
{
  array_init(return_value);

  SPL_LIST_CLASSES(return_value, 0, 0, 0)
}
/* }}} */

static int spl_autoload(zend_string *class_name, zend_string *lc_name, const char *ext, int ext_len) /* {{{ */
{
  char *class_file;
  int class_file_len;
  zval dummy;
  zend_file_handle file_handle;
  zend_op_array *new_op_array;
  zval result;
  int ret;

  class_file_len = (int)spprintf(&class_file, 0, "%s%.*s", ZSTR_VAL(lc_name), ext_len, ext);

#if DEFAULT_SLASH != '\\'
  {
    char *ptr = class_file;
    char *end = ptr + class_file_len;

    while ((ptr = memchr(ptr, '\\', (end - ptr))) != NULL) {
      *ptr = DEFAULT_SLASH;
    }
  }
#endif

  ret = php_stream_open_for_zend_ex(class_file, &file_handle, USE_PATH|STREAM_OPEN_FOR_INCLUDE);

  if (ret == SUCCESS) {
    zend_string *opened_path;
    if (!file_handle.opened_path) {
      file_handle.opened_path = zend_string_init(class_file, class_file_len, 0);
    }
    opened_path = zend_string_copy(file_handle.opened_path);
    ZVAL_NULL(&dummy);
    if (zend_hash_add(&EG(included_files), opened_path, &dummy)) {
      new_op_array = zend_compile_file(&file_handle, ZEND_REQUIRE);
      zend_destroy_file_handle(&file_handle);
    } else {
      new_op_array = NULL;
      zend_file_handle_dtor(&file_handle);
    }
    zend_string_release(opened_path);
    if (new_op_array) {
      ZVAL_UNDEF(&result);
      zend_execute(new_op_array, &result);

      destroy_op_array(new_op_array);
      efree(new_op_array);
      if (!EG(exception)) {
        zval_ptr_dtor(&result);
      }

      efree(class_file);
      return zend_hash_exists(EG(class_table), lc_name);
    }
  }
  efree(class_file);
  return 0;
} /* }}} */

/* {{{ proto void spl_autoload(string class_name [, string file_extensions])
 Default implementation for __autoload() */
PHP_FUNCTION(spl_autoload)
{
  int pos_len, pos1_len;
  char *pos, *pos1;
  zend_string *class_name, *lc_name, *file_exts = SPL_G(autoload_extensions);

  if (zend_parse_parameters(ZEND_NUM_ARGS(), "S|S", &class_name, &file_exts) == FAILURE) {
    RETURN_FALSE;
  }

  if (file_exts == NULL) { /* autoload_extensions is not initialized, set to defaults */
    pos = SPL_DEFAULT_FILE_EXTENSIONS;
    pos_len = sizeof(SPL_DEFAULT_FILE_EXTENSIONS) - 1;
  } else {
    pos = ZSTR_VAL(file_exts);
    pos_len = (int)ZSTR_LEN(file_exts);
  }

  lc_name = zend_string_tolower(class_name);
  while (pos && *pos && !EG(exception)) {
    pos1 = strchr(pos, ',');
    if (pos1) {
      pos1_len = (int)(pos1 - pos);
    } else {
      pos1_len = pos_len;
    }
    if (spl_autoload(class_name, lc_name, pos, pos1_len)) {
      break; /* loaded */
    }
    pos = pos1 ? pos1 + 1 : NULL;
    pos_len = pos1? pos_len - pos1_len - 1 : 0;
  }
  zend_string_free(lc_name);
} /* }}} */

/* {{{ proto string spl_autoload_extensions([string file_extensions])
 Register and return default file extensions for spl_autoload */
PHP_FUNCTION(spl_autoload_extensions)
{
  zend_string *file_exts = NULL;

  if (zend_parse_parameters(ZEND_NUM_ARGS(), "|S", &file_exts) == FAILURE) {
    return;
  }
  if (file_exts) {
    if (SPL_G(autoload_extensions)) {
      zend_string_release(SPL_G(autoload_extensions));
    }
    SPL_G(autoload_extensions) = zend_string_copy(file_exts);
  }

  if (SPL_G(autoload_extensions) == NULL) {
    RETURN_STRINGL(SPL_DEFAULT_FILE_EXTENSIONS, sizeof(SPL_DEFAULT_FILE_EXTENSIONS) - 1);
  } else {
    zend_string_addref(SPL_G(autoload_extensions));
    RETURN_STR(SPL_G(autoload_extensions));
  }
} /* }}} */

typedef struct {
  zend_function *func_ptr;
  zval obj;
  zval closure;
  zend_class_entry *ce;
} autoload_func_info;

static void autoload_func_info_dtor(zval *element)
{
  autoload_func_info *alfi = (autoload_func_info*)Z_PTR_P(element);
  if (!Z_ISUNDEF(alfi->obj)) {
    zval_ptr_dtor(&alfi->obj);
  }
  if (alfi->func_ptr &&
    UNEXPECTED(alfi->func_ptr->common.fn_flags & ZEND_ACC_CALL_VIA_TRAMPOLINE)) {
    zend_string_release(alfi->func_ptr->common.function_name);
    zend_free_trampoline(alfi->func_ptr);
  }
  if (!Z_ISUNDEF(alfi->closure)) {
    zval_ptr_dtor(&alfi->closure);
  }
  efree(alfi);
}

/* {{{ proto void spl_autoload_call(string class_name)
 Try all registerd autoload function to load the requested class */
PHP_FUNCTION(spl_autoload_call)
{
  zval *class_name, retval;
  zend_string *lc_name, *func_name;
  autoload_func_info *alfi;

  if (zend_parse_parameters(ZEND_NUM_ARGS(), "z", &class_name) == FAILURE || Z_TYPE_P(class_name) != IS_STRING) {
    return;
  }

  if (SPL_G(autoload_functions)) {
    HashPosition pos;
    zend_ulong num_idx;
    zend_function *func;
    zend_fcall_info fci;
    zend_fcall_info_cache fcic;
    zend_class_entry *called_scope = zend_get_called_scope(execute_data);
    int l_autoload_running = SPL_G(autoload_running);

    SPL_G(autoload_running) = 1;
    lc_name = zend_string_tolower(Z_STR_P(class_name));

    fci.size = sizeof(fci);
    fci.retval = &retval;
    fci.param_count = 1;
    fci.params = class_name;
    fci.no_separation = 1;

    ZVAL_UNDEF(&fci.function_name); /* Unused */

    zend_hash_internal_pointer_reset_ex(SPL_G(autoload_functions), &pos);
    while (zend_hash_get_current_key_ex(SPL_G(autoload_functions), &func_name, &num_idx, &pos) == HASH_KEY_IS_STRING) {
      alfi = zend_hash_get_current_data_ptr_ex(SPL_G(autoload_functions), &pos);
      func = alfi->func_ptr;
      if (UNEXPECTED(func->common.fn_flags & ZEND_ACC_CALL_VIA_TRAMPOLINE)) {
        func = emalloc(sizeof(zend_op_array));
        memcpy(func, alfi->func_ptr, sizeof(zend_op_array));
        zend_string_addref(func->op_array.function_name);
      }
      ZVAL_UNDEF(&retval);
      fcic.function_handler = func;
      if (Z_ISUNDEF(alfi->obj)) {
        fci.object = NULL;
        fcic.object = NULL;
        fcic.calling_scope = alfi->ce;
        if (alfi->ce &&
            (!called_scope ||
             !instanceof_function(called_scope, alfi->ce))) {
          fcic.called_scope = alfi->ce;
        } else {
          fcic.called_scope = called_scope;
        }
      } else {
        fci.object = Z_OBJ(alfi->obj);
        fcic.object = Z_OBJ(alfi->obj);
        fcic.called_scope = Z_OBJCE(alfi->obj);
      }

      zend_call_function(&fci, &fcic);
      zval_ptr_dtor(&retval);

      if (EG(exception)) {
        break;
      }

      if (pos + 1 == SPL_G(autoload_functions)->nNumUsed ||
          zend_hash_exists(EG(class_table), lc_name)) {
        break;
      }
      zend_hash_move_forward_ex(SPL_G(autoload_functions), &pos);
    }
    zend_string_release(lc_name);
    SPL_G(autoload_running) = l_autoload_running;
  } else {
    /* do not use or overwrite &EG(autoload_func) here */
    zend_call_method_with_1_params(NULL, NULL, NULL, "spl_autoload", NULL, class_name);
  }
} /* }}} */

#define HT_MOVE_TAIL_TO_HEAD(ht)                        \
  do {                                                  \
    Bucket tmp = (ht)->arData[(ht)->nNumUsed-1];        \
    memmove((ht)->arData + 1, (ht)->arData,             \
      sizeof(Bucket) * ((ht)->nNumUsed - 1));           \
    (ht)->arData[0] = tmp;                              \
    zend_hash_rehash(ht);                               \
  } while (0)

/* {{{ proto bool spl_autoload_register([mixed autoload_function [, bool throw [, bool prepend]]])
 Register given function as __autoload() implementation */
PHP_FUNCTION(spl_autoload_register)
{
  zend_string *func_name;
  char *error = NULL;
  zend_string *lc_name;
  zval *zcallable = NULL;
  zend_bool do_throw = 1;
  zend_bool prepend  = 0;
  zend_function *spl_func_ptr;
  autoload_func_info alfi;
  zend_object *obj_ptr;
  zend_fcall_info_cache fcc;

  if (zend_parse_parameters_ex(ZEND_PARSE_PARAMS_QUIET, ZEND_NUM_ARGS(), "|zbb", &zcallable, &do_throw, &prepend) == FAILURE) {
    return;
  }

  if (ZEND_NUM_ARGS()) {
    if (!zend_is_callable_ex(zcallable, NULL, IS_CALLABLE_STRICT, &func_name, &fcc, &error)) {
      alfi.ce = fcc.calling_scope;
      alfi.func_ptr = fcc.function_handler;
      obj_ptr = fcc.object;
      if (Z_TYPE_P(zcallable) == IS_ARRAY) {
        if (!obj_ptr && alfi.func_ptr && !(alfi.func_ptr->common.fn_flags & ZEND_ACC_STATIC)) {
          if (do_throw) {
            zend_throw_exception_ex(spl_ce_LogicException, 0, "Passed array specifies a non static method but no object (%s)", error);
          }
          if (error) {
            efree(error);
          }
          zend_string_release(func_name);
          RETURN_FALSE;
        } else if (do_throw) {
          zend_throw_exception_ex(spl_ce_LogicException, 0, "Passed array does not specify %s %smethod (%s)", alfi.func_ptr ? "a callable" : "an existing", !obj_ptr ? "static " : "", error);
        }
        if (error) {
          efree(error);
        }
        zend_string_release(func_name);
        RETURN_FALSE;
      } else if (Z_TYPE_P(zcallable) == IS_STRING) {
        if (do_throw) {
          zend_throw_exception_ex(spl_ce_LogicException, 0, "Function '%s' not %s (%s)", ZSTR_VAL(func_name), alfi.func_ptr ? "callable" : "found", error);
        }
        if (error) {
          efree(error);
        }
        zend_string_release(func_name);
        RETURN_FALSE;
      } else {
        if (do_throw) {
          zend_throw_exception_ex(spl_ce_LogicException, 0, "Illegal value passed (%s)", error);
        }
        if (error) {
          efree(error);
        }
        zend_string_release(func_name);
        RETURN_FALSE;
      }
    } else if (fcc.function_handler->type == ZEND_INTERNAL_FUNCTION &&
           fcc.function_handler->internal_function.handler == zif_spl_autoload_call) {
      if (do_throw) {
        zend_throw_exception_ex(spl_ce_LogicException, 0, "Function spl_autoload_call() cannot be registered");
      }
      if (error) {
        efree(error);
      }
      zend_string_release(func_name);
      RETURN_FALSE;
    }
    alfi.ce = fcc.calling_scope;
    alfi.func_ptr = fcc.function_handler;
    obj_ptr = fcc.object;
    if (error) {
      efree(error);
    }

    if (Z_TYPE_P(zcallable) == IS_OBJECT) {
      ZVAL_COPY(&alfi.closure, zcallable);

      lc_name = zend_string_alloc(ZSTR_LEN(func_name) + sizeof(uint32_t), 0);
      zend_str_tolower_copy(ZSTR_VAL(lc_name), ZSTR_VAL(func_name), ZSTR_LEN(func_name));
      memcpy(ZSTR_VAL(lc_name) + ZSTR_LEN(func_name), &Z_OBJ_HANDLE_P(zcallable), sizeof(uint32_t));
      ZSTR_VAL(lc_name)[ZSTR_LEN(lc_name)] = '\0';
    } else {
      ZVAL_UNDEF(&alfi.closure);
      /* Skip leading \ */
      if (ZSTR_VAL(func_name)[0] == '\\') {
        lc_name = zend_string_alloc(ZSTR_LEN(func_name) - 1, 0);
        zend_str_tolower_copy(ZSTR_VAL(lc_name), ZSTR_VAL(func_name) + 1, ZSTR_LEN(func_name) - 1);
      } else {
        lc_name = zend_string_tolower(func_name);
      }
    }
    zend_string_release(func_name);

    if (SPL_G(autoload_functions) && zend_hash_exists(SPL_G(autoload_functions), lc_name)) {
      if (!Z_ISUNDEF(alfi.closure)) {
        Z_DELREF_P(&alfi.closure);
      }
      goto skip;
    }

    if (obj_ptr && !(alfi.func_ptr->common.fn_flags & ZEND_ACC_STATIC)) {
      /* add object id to the hash to ensure uniqueness, for more reference look at bug #40091 */
      lc_name = zend_string_extend(lc_name, ZSTR_LEN(lc_name) + sizeof(uint32_t), 0);
      memcpy(ZSTR_VAL(lc_name) + ZSTR_LEN(lc_name) - sizeof(uint32_t), &obj_ptr->handle, sizeof(uint32_t));
      ZSTR_VAL(lc_name)[ZSTR_LEN(lc_name)] = '\0';
      ZVAL_OBJ(&alfi.obj, obj_ptr);
      Z_ADDREF(alfi.obj);
    } else {
      ZVAL_UNDEF(&alfi.obj);
    }

    if (!SPL_G(autoload_functions)) {
      ALLOC_HASHTABLE(SPL_G(autoload_functions));
      zend_hash_init(SPL_G(autoload_functions), 1, NULL, autoload_func_info_dtor, 0);
    }

    spl_func_ptr = zend_hash_str_find_ptr(EG(function_table), "spl_autoload", sizeof("spl_autoload") - 1);

    if (EG(autoload_func) == spl_func_ptr) { /* registered already, so we insert that first */
      autoload_func_info spl_alfi;

      spl_alfi.func_ptr = spl_func_ptr;
      ZVAL_UNDEF(&spl_alfi.obj);
      ZVAL_UNDEF(&spl_alfi.closure);
      spl_alfi.ce = NULL;
      zend_hash_str_add_mem(SPL_G(autoload_functions), "spl_autoload", sizeof("spl_autoload") - 1,
          &spl_alfi, sizeof(autoload_func_info));
      if (prepend && SPL_G(autoload_functions)->nNumOfElements > 1) {
        /* Move the newly created element to the head of the hashtable */
        HT_MOVE_TAIL_TO_HEAD(SPL_G(autoload_functions));
      }
    }

    if (UNEXPECTED(alfi.func_ptr == &EG(trampoline))) {
      zend_function *copy = emalloc(sizeof(zend_op_array));

      memcpy(copy, alfi.func_ptr, sizeof(zend_op_array));
      alfi.func_ptr->common.function_name = NULL;
      alfi.func_ptr = copy;
    }
    if (zend_hash_add_mem(SPL_G(autoload_functions), lc_name, &alfi, sizeof(autoload_func_info)) == NULL) {
      if (obj_ptr && !(alfi.func_ptr->common.fn_flags & ZEND_ACC_STATIC)) {
        Z_DELREF(alfi.obj);
      }
      if (!Z_ISUNDEF(alfi.closure)) {
        Z_DELREF(alfi.closure);
      }
      if (UNEXPECTED(alfi.func_ptr->common.fn_flags & ZEND_ACC_CALL_VIA_TRAMPOLINE)) {
        zend_string_release(alfi.func_ptr->common.function_name);
        zend_free_trampoline(alfi.func_ptr);
      }
    }
    if (prepend && SPL_G(autoload_functions)->nNumOfElements > 1) {
      /* Move the newly created element to the head of the hashtable */
      HT_MOVE_TAIL_TO_HEAD(SPL_G(autoload_functions));
    }
skip:
    zend_string_release(lc_name);
  }

  if (SPL_G(autoload_functions)) {
    EG(autoload_func) = zend_hash_str_find_ptr(EG(function_table), "spl_autoload_call", sizeof("spl_autoload_call") - 1);
  } else {
    EG(autoload_func) = zend_hash_str_find_ptr(EG(function_table), "spl_autoload", sizeof("spl_autoload") - 1);
  }

  RETURN_TRUE;
} /* }}} */

/* {{{ proto bool spl_autoload_unregister(mixed autoload_function)
 Unregister given function as __autoload() implementation */
PHP_FUNCTION(spl_autoload_unregister)
{
  zend_string *func_name = NULL;
  char *error = NULL;
  zend_string *lc_name;
  zval *zcallable;
  int success = FAILURE;
  zend_function *spl_func_ptr;
  zend_object *obj_ptr;
  zend_fcall_info_cache fcc;

  if (zend_parse_parameters(ZEND_NUM_ARGS(), "z", &zcallable) == FAILURE) {
    return;
  }

  if (!zend_is_callable_ex(zcallable, NULL, IS_CALLABLE_CHECK_SYNTAX_ONLY, &func_name, &fcc, &error)) {
    zend_throw_exception_ex(spl_ce_LogicException, 0, "Unable to unregister invalid function (%s)", error);
    if (error) {
      efree(error);
    }
    if (func_name) {
      zend_string_release(func_name);
    }
    RETURN_FALSE;
  }
  obj_ptr = fcc.object;
  if (error) {
    efree(error);
  }

  if (Z_TYPE_P(zcallable) == IS_OBJECT) {
    lc_name = zend_string_alloc(ZSTR_LEN(func_name) + sizeof(uint32_t), 0);
    zend_str_tolower_copy(ZSTR_VAL(lc_name), ZSTR_VAL(func_name), ZSTR_LEN(func_name));
    memcpy(ZSTR_VAL(lc_name) + ZSTR_LEN(func_name), &Z_OBJ_HANDLE_P(zcallable), sizeof(uint32_t));
    ZSTR_VAL(lc_name)[ZSTR_LEN(lc_name)] = '\0';
  } else {
    /* Skip leading \ */
    if (ZSTR_VAL(func_name)[0] == '\\') {
      lc_name = zend_string_alloc(ZSTR_LEN(func_name) - 1, 0);
      zend_str_tolower_copy(ZSTR_VAL(lc_name), ZSTR_VAL(func_name) + 1, ZSTR_LEN(func_name) - 1);
    } else {
      lc_name = zend_string_tolower(func_name);
    }
  }
  zend_string_release(func_name);

  if (SPL_G(autoload_functions)) {
    if (ZSTR_LEN(lc_name) == sizeof("spl_autoload_call") - 1 && !strcmp(ZSTR_VAL(lc_name), "spl_autoload_call")) {
      /* remove all */
      if (!SPL_G(autoload_running)) {
        zend_hash_destroy(SPL_G(autoload_functions));
        FREE_HASHTABLE(SPL_G(autoload_functions));
        SPL_G(autoload_functions) = NULL;
        EG(autoload_func) = NULL;
      } else {
        zend_hash_clean(SPL_G(autoload_functions));
      }
      success = SUCCESS;
    } else {
      /* remove specific */
      success = zend_hash_del(SPL_G(autoload_functions), lc_name);
      if (success != SUCCESS && obj_ptr) {
        lc_name = zend_string_extend(lc_name, ZSTR_LEN(lc_name) + sizeof(uint32_t), 0);
        memcpy(ZSTR_VAL(lc_name) + ZSTR_LEN(lc_name) - sizeof(uint32_t), &obj_ptr->handle, sizeof(uint32_t));
        ZSTR_VAL(lc_name)[ZSTR_LEN(lc_name)] = '\0';
        success = zend_hash_del(SPL_G(autoload_functions), lc_name);
      }
    }
  } else if (ZSTR_LEN(lc_name) == sizeof("spl_autoload")-1 && !strcmp(ZSTR_VAL(lc_name), "spl_autoload")) {
    /* register single spl_autoload() */
    spl_func_ptr = zend_hash_str_find_ptr(EG(function_table), "spl_autoload", sizeof("spl_autoload") - 1);

    if (EG(autoload_func) == spl_func_ptr) {
      success = SUCCESS;
      EG(autoload_func) = NULL;
    }
  }

  zend_string_release(lc_name);
  RETURN_BOOL(success == SUCCESS);
} /* }}} */

/* {{{ proto false|array spl_autoload_functions()
 Return all registered __autoload() functionns */
PHP_FUNCTION(spl_autoload_functions)
{
  zend_function *fptr;
  autoload_func_info *alfi;

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  if (!EG(autoload_func)) {
    if ((fptr = zend_hash_str_find_ptr(EG(function_table), ZEND_AUTOLOAD_FUNC_NAME, sizeof(ZEND_AUTOLOAD_FUNC_NAME) - 1))) {
      array_init(return_value);
      add_next_index_stringl(return_value, ZEND_AUTOLOAD_FUNC_NAME, sizeof(ZEND_AUTOLOAD_FUNC_NAME)-1);
      return;
    }
    RETURN_FALSE;
  }

  fptr = zend_hash_str_find_ptr(EG(function_table), "spl_autoload_call", sizeof("spl_autoload_call") - 1);

  if (EG(autoload_func) == fptr) {
    zend_string *key;
    array_init(return_value);
    ZEND_HASH_FOREACH_STR_KEY_PTR(SPL_G(autoload_functions), key, alfi) {
      if (!Z_ISUNDEF(alfi->closure)) {
        Z_ADDREF(alfi->closure);
        add_next_index_zval(return_value, &alfi->closure);
      } else if (alfi->func_ptr->common.scope) {
        zval tmp;

        array_init(&tmp);
        if (!Z_ISUNDEF(alfi->obj)) {
          Z_ADDREF(alfi->obj);
          add_next_index_zval(&tmp, &alfi->obj);
        } else {
          add_next_index_str(&tmp, zend_string_copy(alfi->ce->name));
        }
        add_next_index_str(&tmp, zend_string_copy(alfi->func_ptr->common.function_name));
        add_next_index_zval(return_value, &tmp);
      } else {
        if (strncmp(ZSTR_VAL(alfi->func_ptr->common.function_name), "__lambda_func", sizeof("__lambda_func") - 1)) {
          add_next_index_str(return_value, zend_string_copy(alfi->func_ptr->common.function_name));
        } else {
          add_next_index_str(return_value, zend_string_copy(key));
        }
      }
    } ZEND_HASH_FOREACH_END();
    return;
  }

  array_init(return_value);
  add_next_index_str(return_value, zend_string_copy(EG(autoload_func)->common.function_name));
} /* }}} */

/* {{{ proto string spl_object_hash(object obj)
 Return hash id for given object */
PHP_FUNCTION(spl_object_hash)
{
  zval *obj;

  if (zend_parse_parameters(ZEND_NUM_ARGS(), "o", &obj) == FAILURE) {
    return;
  }

  RETURN_NEW_STR(php_spl_object_hash(obj));
}
/* }}} */

/* {{{ proto int spl_object_id(object obj)
 Returns the integer object handle for the given object */
PHP_FUNCTION(spl_object_id)
{
  zval *obj;

  ZEND_PARSE_PARAMETERS_START(1, 1)
    Z_PARAM_OBJECT(obj)
  ZEND_PARSE_PARAMETERS_END();

  RETURN_LONG((zend_long)Z_OBJ_HANDLE_P(obj));
}
/* }}} */

PHPAPI zend_string *php_spl_object_hash(zval *obj) /* {{{*/
{
  intptr_t hash_handle, hash_handlers;

  if (!SPL_G(hash_mask_init)) {
    SPL_G(hash_mask_handle)   = (intptr_t)(php_mt_rand() >> 1);
    SPL_G(hash_mask_handlers) = (intptr_t)(php_mt_rand() >> 1);
    SPL_G(hash_mask_init) = 1;
  }

  hash_handle   = SPL_G(hash_mask_handle)^(intptr_t)Z_OBJ_HANDLE_P(obj);
  hash_handlers = SPL_G(hash_mask_handlers);

  return strpprintf(32, "%016zx%016zx", hash_handle, hash_handlers);
}
/* }}} */

int spl_build_class_list_string(zval *entry, char **list) /* {{{ */
{
  char *res;

  spprintf(&res, 0, "%s, %s", *list, Z_STRVAL_P(entry));
  efree(*list);
  *list = res;
  return ZEND_HASH_APPLY_KEEP;
} /* }}} */

/* {{{ PHP_MINFO(spl)
 */
PHP_MINFO_FUNCTION(spl)
{
  zval list;
  char *strg;

  php_info_print_table_start();
  php_info_print_table_header(2, "SPL support",        "enabled");

  array_init(&list);
  SPL_LIST_CLASSES(&list, 0, 1, ZEND_ACC_INTERFACE)
  strg = estrdup("");
  zend_hash_apply_with_argument(Z_ARRVAL_P(&list), (apply_func_arg_t)spl_build_class_list_string, &strg);
  zval_dtor(&list);
  php_info_print_table_row(2, "Interfaces", strg + 2);
  efree(strg);

  array_init(&list);
  SPL_LIST_CLASSES(&list, 0, -1, ZEND_ACC_INTERFACE)
  strg = estrdup("");
  zend_hash_apply_with_argument(Z_ARRVAL_P(&list), (apply_func_arg_t)spl_build_class_list_string, &strg);
  zval_dtor(&list);
  php_info_print_table_row(2, "Classes", strg + 2);
  efree(strg);

  php_info_print_table_end();
}
/* }}} */

/* {{{ arginfo */
ZEND_BEGIN_ARG_INFO_EX(arginfo_iterator_to_array, 0, 0, 1)
  ZEND_ARG_OBJ_INFO(0, iterator, Traversable, 0)
  ZEND_ARG_INFO(0, use_keys)
ZEND_END_ARG_INFO();

ZEND_BEGIN_ARG_INFO(arginfo_iterator, 0)
  ZEND_ARG_OBJ_INFO(0, iterator, Traversable, 0)
ZEND_END_ARG_INFO();

ZEND_BEGIN_ARG_INFO_EX(arginfo_iterator_apply, 0, 0, 2)
  ZEND_ARG_OBJ_INFO(0, iterator, Traversable, 0)
  ZEND_ARG_INFO(0, function)
  ZEND_ARG_ARRAY_INFO(0, args, 1)
ZEND_END_ARG_INFO();

ZEND_BEGIN_ARG_INFO_EX(arginfo_class_parents, 0, 0, 1)
  ZEND_ARG_INFO(0, instance)
  ZEND_ARG_INFO(0, autoload)
ZEND_END_ARG_INFO()

ZEND_BEGIN_ARG_INFO_EX(arginfo_class_implements, 0, 0, 1)
  ZEND_ARG_INFO(0, what)
  ZEND_ARG_INFO(0, autoload)
ZEND_END_ARG_INFO()

ZEND_BEGIN_ARG_INFO_EX(arginfo_class_uses, 0, 0, 1)
  ZEND_ARG_INFO(0, what)
  ZEND_ARG_INFO(0, autoload)
ZEND_END_ARG_INFO()


ZEND_BEGIN_ARG_INFO(arginfo_spl_classes, 0)
ZEND_END_ARG_INFO()

ZEND_BEGIN_ARG_INFO(arginfo_spl_autoload_functions, 0)
ZEND_END_ARG_INFO()

ZEND_BEGIN_ARG_INFO_EX(arginfo_spl_autoload, 0, 0, 1)
  ZEND_ARG_INFO(0, class_name)
  ZEND_ARG_INFO(0, file_extensions)
ZEND_END_ARG_INFO()

ZEND_BEGIN_ARG_INFO_EX(arginfo_spl_autoload_extensions, 0, 0, 0)
  ZEND_ARG_INFO(0, file_extensions)
ZEND_END_ARG_INFO()

ZEND_BEGIN_ARG_INFO_EX(arginfo_spl_autoload_call, 0, 0, 1)
  ZEND_ARG_INFO(0, class_name)
ZEND_END_ARG_INFO()

ZEND_BEGIN_ARG_INFO_EX(arginfo_spl_autoload_register, 0, 0, 0)
  ZEND_ARG_INFO(0, autoload_function)
  ZEND_ARG_INFO(0, throw)
  ZEND_ARG_INFO(0, prepend)
ZEND_END_ARG_INFO()

ZEND_BEGIN_ARG_INFO_EX(arginfo_spl_autoload_unregister, 0, 0, 1)
  ZEND_ARG_INFO(0, autoload_function)
ZEND_END_ARG_INFO()

ZEND_BEGIN_ARG_INFO_EX(arginfo_spl_object_hash, 0, 0, 1)
  ZEND_ARG_INFO(0, obj)
ZEND_END_ARG_INFO()

ZEND_BEGIN_ARG_INFO_EX(arginfo_spl_object_id, 0, 0, 1)
  ZEND_ARG_INFO(0, obj)
ZEND_END_ARG_INFO()
/* }}} */

/* {{{ spl_functions
 */
static const zend_function_entry spl_functions[] = {
  PHP_FE(spl_classes,             arginfo_spl_classes)
  PHP_FE(spl_autoload,            arginfo_spl_autoload)
  PHP_FE(spl_autoload_extensions, arginfo_spl_autoload_extensions)
  PHP_FE(spl_autoload_register,   arginfo_spl_autoload_register)
  PHP_FE(spl_autoload_unregister, arginfo_spl_autoload_unregister)
  PHP_FE(spl_autoload_functions,  arginfo_spl_autoload_functions)
  PHP_FE(spl_autoload_call,       arginfo_spl_autoload_call)
  PHP_FE(class_parents,           arginfo_class_parents)
  PHP_FE(class_implements,        arginfo_class_implements)
  PHP_FE(class_uses,              arginfo_class_uses)
  PHP_FE(spl_object_hash,         arginfo_spl_object_hash)
  PHP_FE(spl_object_id,           arginfo_spl_object_id)
#ifdef SPL_ITERATORS_H
  PHP_FE(iterator_to_array,       arginfo_iterator_to_array)
  PHP_FE(iterator_count,          arginfo_iterator)
  PHP_FE(iterator_apply,          arginfo_iterator_apply)
#endif /* SPL_ITERATORS_H */
  PHP_FE_END
};
/* }}} */

/* {{{ PHP_MINIT_FUNCTION(spl)
 */
PHP_MINIT_FUNCTION(spl)
{
  PHP_MINIT(spl_exceptions)(INIT_FUNC_ARGS_PASSTHRU);
  PHP_MINIT(spl_iterators)(INIT_FUNC_ARGS_PASSTHRU);
  PHP_MINIT(spl_array)(INIT_FUNC_ARGS_PASSTHRU);
  PHP_MINIT(spl_directory)(INIT_FUNC_ARGS_PASSTHRU);
  PHP_MINIT(spl_dllist)(INIT_FUNC_ARGS_PASSTHRU);
  PHP_MINIT(spl_heap)(INIT_FUNC_ARGS_PASSTHRU);
  PHP_MINIT(spl_fixedarray)(INIT_FUNC_ARGS_PASSTHRU);
  PHP_MINIT(spl_observer)(INIT_FUNC_ARGS_PASSTHRU);

  return SUCCESS;
}
/* }}} */

PHP_RINIT_FUNCTION(spl) /* {{{ */
{
  SPL_G(autoload_extensions) = NULL;
  SPL_G(autoload_functions) = NULL;
  SPL_G(hash_mask_init) = 0;
  return SUCCESS;
} /* }}} */

PHP_RSHUTDOWN_FUNCTION(spl) /* {{{ */
{
  if (SPL_G(autoload_extensions)) {
    zend_string_release(SPL_G(autoload_extensions));
    SPL_G(autoload_extensions) = NULL;
  }
  if (SPL_G(autoload_functions)) {
    zend_hash_destroy(SPL_G(autoload_functions));
    FREE_HASHTABLE(SPL_G(autoload_functions));
    SPL_G(autoload_functions) = NULL;
  }
  if (SPL_G(hash_mask_init)) {
    SPL_G(hash_mask_init) = 0;
  }
  return SUCCESS;
} /* }}} */

/* {{{ spl_module_entry
 */
zend_module_entry spl_module_entry = {
  STANDARD_MODULE_HEADER,
  "SPL",
  spl_functions,
  PHP_MINIT(spl),
  NULL,
  PHP_RINIT(spl),
  PHP_RSHUTDOWN(spl),
  PHP_MINFO(spl),
  PHP_SPL_VERSION,
  PHP_MODULE_GLOBALS(spl),
  PHP_GINIT(spl),
  NULL,
  NULL,
  STANDARD_MODULE_PROPERTIES_EX
};
/* }}} */

/*
 * Local variables:
 * tab-width: 4
 * c-basic-offset: 4
 * End:
 * vim600: fdm=marker
 * vim: noet sw=4 ts=4
 */
```

```c
/*
   +----------------------------------------------------------------------+
   | PHP Version 7                                                        |
   +----------------------------------------------------------------------+
   | Copyright (c) 1997-2018 The PHP Group                                |
   +----------------------------------------------------------------------+
   | This source file is subject to version 3.01 of the PHP license,      |
   | that is bundled with this package in the file LICENSE, and is        |
   | available through the world-wide-web at the following url:           |
   | http://www.php.net/license/3_01.txt                                  |
   | If you did not receive a copy of the PHP license and are unable to   |
   | obtain it through the world-wide-web, please send a note to          |
   | license@php.net so we can mail you a copy immediately.               |
   +----------------------------------------------------------------------+
   | Authors: Marcus Boerger <helly@php.net>                              |
   +----------------------------------------------------------------------+
 */

/* $Id$ */

#ifdef HAVE_CONFIG_H
# include "config.h"
#endif

#include "php.h"
#include "php_ini.h"
#include "ext/standard/info.h"
#include "zend_interfaces.h"
#include "zend_exceptions.h"

#include "php_spl.h"
#include "spl_functions.h"
#include "spl_engine.h"
#include "spl_exceptions.h"

PHPAPI zend_class_entry *spl_ce_LogicException;
PHPAPI zend_class_entry *spl_ce_BadFunctionCallException;
PHPAPI zend_class_entry *spl_ce_BadMethodCallException;
PHPAPI zend_class_entry *spl_ce_DomainException;
PHPAPI zend_class_entry *spl_ce_InvalidArgumentException;
PHPAPI zend_class_entry *spl_ce_LengthException;
PHPAPI zend_class_entry *spl_ce_OutOfRangeException;
PHPAPI zend_class_entry *spl_ce_RuntimeException;
PHPAPI zend_class_entry *spl_ce_OutOfBoundsException;
PHPAPI zend_class_entry *spl_ce_OverflowException;
PHPAPI zend_class_entry *spl_ce_RangeException;
PHPAPI zend_class_entry *spl_ce_UnderflowException;
PHPAPI zend_class_entry *spl_ce_UnexpectedValueException;

#define spl_ce_Exception zend_ce_exception

/* {{{ PHP_MINIT_FUNCTION(spl_exceptions) */
PHP_MINIT_FUNCTION(spl_exceptions)
{
    REGISTER_SPL_SUB_CLASS_EX(LogicException,           Exception,        NULL, NULL);
    REGISTER_SPL_SUB_CLASS_EX(BadFunctionCallException, LogicException,   NULL, NULL);
    REGISTER_SPL_SUB_CLASS_EX(BadMethodCallException,   BadFunctionCallException,   NULL, NULL);
    REGISTER_SPL_SUB_CLASS_EX(DomainException,          LogicException,   NULL, NULL);
    REGISTER_SPL_SUB_CLASS_EX(InvalidArgumentException, LogicException,   NULL, NULL);
    REGISTER_SPL_SUB_CLASS_EX(LengthException,          LogicException,   NULL, NULL);
    REGISTER_SPL_SUB_CLASS_EX(OutOfRangeException,      LogicException,   NULL, NULL);

    REGISTER_SPL_SUB_CLASS_EX(RuntimeException,         Exception,        NULL, NULL);
    REGISTER_SPL_SUB_CLASS_EX(OutOfBoundsException,     RuntimeException, NULL, NULL);
    REGISTER_SPL_SUB_CLASS_EX(OverflowException,        RuntimeException, NULL, NULL);
    REGISTER_SPL_SUB_CLASS_EX(RangeException,           RuntimeException, NULL, NULL);
    REGISTER_SPL_SUB_CLASS_EX(UnderflowException,       RuntimeException, NULL, NULL);
    REGISTER_SPL_SUB_CLASS_EX(UnexpectedValueException, RuntimeException, NULL, NULL);

  return SUCCESS;
}
/* }}} */

/*
 * Local variables:
 * tab-width: 4
 * c-basic-offset: 4
 * End:
 * vim600: fdm=marker
 * vim: noet sw=4 ts=4
 */
```

```c
/*
   +----------------------------------------------------------------------+
   | PHP Version 7                                                        |
   +----------------------------------------------------------------------+
   | Copyright (c) 1997-2018 The PHP Group                                |
   +----------------------------------------------------------------------+
   | This source file is subject to version 3.01 of the PHP license,      |
   | that is bundled with this package in the file LICENSE, and is        |
   | available through the world-wide-web at the following url:          |
   | http://www.php.net/license/3_01.txt                                  |
   | If you did not receive a copy of the PHP license and are unable to   |
   | obtain it through the world-wide-web, please send a note to          |
   | license@php.net so we can mail you a copy immediately.               |
   +----------------------------------------------------------------------+
   | Authors: Marcus Boerger <helly@php.net>                              |
   +----------------------------------------------------------------------+
 */

/* $Id$ */

#ifndef SPL_OBSERVER_H
#define SPL_OBSERVER_H

#include "php.h"
#include "php_spl.h"

extern PHPAPI zend_class_entry *spl_ce_SplObserver;
extern PHPAPI zend_class_entry *spl_ce_SplSubject;
extern PHPAPI zend_class_entry *spl_ce_SplObjectStorage;
extern PHPAPI zend_class_entry *spl_ce_MultipleIterator;

PHP_MINIT_FUNCTION(spl_observer);

#endif /* SPL_OBSERVER_H */

/*
 * Local Variables:
 * c-basic-offset: 4
 * tab-width: 4
 * End:
 * vim600: fdm=marker
 * vim: noet sw=4 ts=4
 */
```

```c
/*
   +----------------------------------------------------------------------+
   | PHP Version 7                                                        |
   +----------------------------------------------------------------------+
   | Copyright (c) 1997-2018 The PHP Group                                |
   +----------------------------------------------------------------------+
   | This source file is subject to version 3.01 of the PHP license,      |
   | that is bundled with this package in the file LICENSE, and is        |
   | available through the world-wide-web at the following url:           |
   | http://www.php.net/license/3_01.txt                                  |
   | If you did not receive a copy of the PHP license and are unable to   |
   | obtain it through the world-wide-web, please send a note to          |
   | license@php.net so we can mail you a copy immediately.               |
   +----------------------------------------------------------------------+
   | Authors: Marcus Boerger <helly@php.net>                              |
   +----------------------------------------------------------------------+
 */

/* $Id$ */

#ifndef SPL_EXCEPTIONS_H
#define SPL_EXCEPTIONS_H

#include "php.h"
#include "php_spl.h"

extern PHPAPI zend_class_entry *spl_ce_LogicException;
extern PHPAPI zend_class_entry *spl_ce_BadFunctionCallException;
extern PHPAPI zend_class_entry *spl_ce_BadMethodCallException;
extern PHPAPI zend_class_entry *spl_ce_DomainException;
extern PHPAPI zend_class_entry *spl_ce_InvalidArgumentException;
extern PHPAPI zend_class_entry *spl_ce_LengthException;
extern PHPAPI zend_class_entry *spl_ce_OutOfRangeException;

extern PHPAPI zend_class_entry *spl_ce_RuntimeException;
extern PHPAPI zend_class_entry *spl_ce_OutOfBoundsException;
extern PHPAPI zend_class_entry *spl_ce_OverflowException;
extern PHPAPI zend_class_entry *spl_ce_RangeException;
extern PHPAPI zend_class_entry *spl_ce_UnderflowException;
extern PHPAPI zend_class_entry *spl_ce_UnexpectedValueException;

PHP_MINIT_FUNCTION(spl_exceptions);

#endif /* SPL_EXCEPTIONS_H */

/*
 * Local Variables:
 * c-basic-offset: 4
 * tab-width: 4
 * End:
 * vim600: fdm=marker
 * vim: noet sw=4 ts=4
 */
```

```c
/*
   +----------------------------------------------------------------------+
   | PHP Version 7                                                        |
   +----------------------------------------------------------------------+
   | Copyright (c) 1997-2018 The PHP Group                                |
   +----------------------------------------------------------------------+
   | This source file is subject to version 3.01 of the PHP license,      |
   | that is bundled with this package in the file LICENSE, and is        |
   | available through the world-wide-web at the following url:           |
   | http://www.php.net/license/3_01.txt                                  |
   | If you did not receive a copy of the PHP license and are unable to   |
   | obtain it through the world-wide-web, please send a note to          |
   | license@php.net so we can mail you a copy immediately.               |
   +----------------------------------------------------------------------+
   | Authors: Marcus Boerger <helly@php.net>                              |
   +----------------------------------------------------------------------+
 */

/* $Id$ */

#ifdef HAVE_CONFIG_H
  #include "config.h"
#endif

#include "php.h"
#include "php_ini.h"
#include "ext/standard/info.h"
#include "php_spl.h"

/* {{{ spl_register_interface */
void spl_register_interface(zend_class_entry ** ppce, char * class_name, const zend_function_entry * functions)
{
  zend_class_entry ce;

  INIT_CLASS_ENTRY_EX(ce, class_name, strlen(class_name), functions);
  *ppce = zend_register_internal_interface(&ce);
}
/* }}} */

/* {{{ spl_register_std_class */
PHPAPI void spl_register_std_class(zend_class_entry ** ppce, char * class_name, void * obj_ctor, const zend_function_entry * function_list)
{
  zend_class_entry ce;

  INIT_CLASS_ENTRY_EX(ce, class_name, strlen(class_name), function_list);
  *ppce = zend_register_internal_class(&ce);

  /* entries changed by initialize */
  if (obj_ctor) {
    (*ppce)->create_object = obj_ctor;
  }
}
/* }}} */

/* {{{ spl_register_sub_class */
PHPAPI void spl_register_sub_class(zend_class_entry ** ppce, zend_class_entry * parent_ce, char * class_name, void *obj_ctor, const zend_function_entry * function_list)
{
  zend_class_entry ce;

  INIT_CLASS_ENTRY_EX(ce, class_name, strlen(class_name), function_list);
  *ppce = zend_register_internal_class_ex(&ce, parent_ce);

  /* entries changed by initialize */
  if (obj_ctor) {
    (*ppce)->create_object = obj_ctor;
  } else {
    (*ppce)->create_object = parent_ce->create_object;
  }
}
/* }}} */

/* {{{ spl_register_property */
void spl_register_property( zend_class_entry * class_entry, char *prop_name, int prop_name_len, int prop_flags)
{
  zend_declare_property_null(class_entry, prop_name, prop_name_len, prop_flags);
}
/* }}} */

/* {{{ spl_add_class_name */
void spl_add_class_name(zval *list, zend_class_entry *pce, int allow, int ce_flags)
{
  if (!allow || (allow > 0 && pce->ce_flags & ce_flags) || (allow < 0 && !(pce->ce_flags & ce_flags))) {
    zval *tmp;

    if ((tmp = zend_hash_find(Z_ARRVAL_P(list), pce->name)) == NULL) {
      zval t;
      ZVAL_STR_COPY(&t, pce->name);
      zend_hash_add(Z_ARRVAL_P(list), pce->name, &t);
    }
  }
}
/* }}} */

/* {{{ spl_add_interfaces */
void spl_add_interfaces(zval *list, zend_class_entry * pce, int allow, int ce_flags)
{
  uint32_t num_interfaces;

  for (num_interfaces = 0; num_interfaces < pce->num_interfaces; num_interfaces++) {
    spl_add_class_name(list, pce->interfaces[num_interfaces], allow, ce_flags);
  }
}
/* }}} */

/* {{{ spl_add_traits */
void spl_add_traits(zval *list, zend_class_entry * pce, int allow, int ce_flags)
{
  uint32_t num_traits;

  for (num_traits = 0; num_traits < pce->num_traits; num_traits++) {
    spl_add_class_name(list, pce->traits[num_traits], allow, ce_flags);
  }
}
/* }}} */


/* {{{ spl_add_classes */
int spl_add_classes(zend_class_entry *pce, zval *list, int sub, int allow, int ce_flags)
{
  if (!pce) {
    return 0;
  }
  spl_add_class_name(list, pce, allow, ce_flags);
  if (sub) {
    spl_add_interfaces(list, pce, allow, ce_flags);
    while (pce->parent) {
      pce = pce->parent;
      spl_add_classes(pce, list, sub, allow, ce_flags);
    }
  }
  return 0;
}
/* }}} */

zend_string * spl_gen_private_prop_name(zend_class_entry *ce, char *prop_name, int prop_len) /* {{{ */
{
  return zend_mangle_property_name(ZSTR_VAL(ce->name), ZSTR_LEN(ce->name), prop_name, prop_len, 0);
}
/* }}} */

/*
 * Local variables:
 * tab-width: 4
 * c-basic-offset: 4
 * End:
 * vim600: fdm=marker
 * vim: noet sw=4 ts=4
 */
```

```c
/*
   +----------------------------------------------------------------------+
   | PHP Version 7                                                        |
   +----------------------------------------------------------------------+
   | Copyright (c) 1997-2018 The PHP Group                                |
   +----------------------------------------------------------------------+
   | This source file is subject to version 3.01 of the PHP license,      |
   | that is bundled with this package in the file LICENSE, and is        |
   | available through the world-wide-web at the following url:           |
   | http://www.php.net/license/3_01.txt                                  |
   | If you did not receive a copy of the PHP license and are unable to   |
   | obtain it through the world-wide-web, please send a note to          |
   | license@php.net so we can mail you a copy immediately.               |
   +----------------------------------------------------------------------+
   | Authors: Marcus Boerger <helly@php.net>                              |
   +----------------------------------------------------------------------+
 */

/* $Id$ */

#ifndef SPL_DIRECTORY_H
#define SPL_DIRECTORY_H

#include "php.h"
#include "php_spl.h"

extern PHPAPI zend_class_entry *spl_ce_SplFileInfo;
extern PHPAPI zend_class_entry *spl_ce_DirectoryIterator;
extern PHPAPI zend_class_entry *spl_ce_FilesystemIterator;
extern PHPAPI zend_class_entry *spl_ce_RecursiveDirectoryIterator;
extern PHPAPI zend_class_entry *spl_ce_GlobIterator;
extern PHPAPI zend_class_entry *spl_ce_SplFileObject;
extern PHPAPI zend_class_entry *spl_ce_SplTempFileObject;

PHP_MINIT_FUNCTION(spl_directory);

typedef enum {
  SPL_FS_INFO, /* must be 0 */
  SPL_FS_DIR,
  SPL_FS_FILE
} SPL_FS_OBJ_TYPE;

typedef struct _spl_filesystem_object  spl_filesystem_object;

typedef void (*spl_foreign_dtor_t)(spl_filesystem_object *object);
typedef void (*spl_foreign_clone_t)(spl_filesystem_object *src, spl_filesystem_object *dst);

PHPAPI char* spl_filesystem_object_get_path(spl_filesystem_object *intern, size_t *len);

typedef struct _spl_other_handler {
  spl_foreign_dtor_t     dtor;
  spl_foreign_clone_t    clone;
} spl_other_handler;

/* define an overloaded iterator structure */
typedef struct {
  zend_object_iterator  intern;
  zval                  current;
  void                 *object;
} spl_filesystem_iterator;

struct _spl_filesystem_object {
  void               *oth;
  const spl_other_handler  *oth_handler;
  char               *_path;
  size_t             _path_len;
  char               *orig_path;
  char               *file_name;
  size_t             file_name_len;
  SPL_FS_OBJ_TYPE    type;
  zend_long               flags;
  zend_class_entry   *file_class;
  zend_class_entry   *info_class;
  union {
    struct {
      php_stream         *dirp;
      php_stream_dirent  entry;
      char               *sub_path;
      size_t             sub_path_len;
      int                index;
      int                is_recursive;
      zend_function      *func_rewind;
      zend_function      *func_next;
      zend_function      *func_valid;
    } dir;
    struct {
      php_stream         *stream;
      php_stream_context *context;
      zval               *zcontext;
      char               *open_mode;
      size_t             open_mode_len;
      zval               current_zval;
      char               *current_line;
      size_t             current_line_len;
      size_t             max_line_len;
      zend_long               current_line_num;
      zval               zresource;
      zend_function      *func_getCurr;
      char               delimiter;
      char               enclosure;
      char               escape;
    } file;
  } u;
  zend_object        std;
};

static inline spl_filesystem_object *spl_filesystem_from_obj(zend_object *obj) /* {{{ */ {
  return (spl_filesystem_object*)((char*)(obj) - XtOffsetOf(spl_filesystem_object, std));
}
/* }}} */

#define Z_SPLFILESYSTEM_P(zv)  spl_filesystem_from_obj(Z_OBJ_P((zv)))

static inline spl_filesystem_iterator* spl_filesystem_object_to_iterator(spl_filesystem_object *obj)
{
  spl_filesystem_iterator    *it;

  it = ecalloc(1, sizeof(spl_filesystem_iterator));
  it->object = (void *)obj;
  zend_iterator_init(&it->intern);
  return it;
}

static inline spl_filesystem_object* spl_filesystem_iterator_to_object(spl_filesystem_iterator *it)
{
  return (spl_filesystem_object*)it->object;
}

#define SPL_FILE_OBJECT_DROP_NEW_LINE      0x00000001 /* drop new lines */
#define SPL_FILE_OBJECT_READ_AHEAD         0x00000002 /* read on rewind/next */
#define SPL_FILE_OBJECT_SKIP_EMPTY         0x00000004 /* skip empty lines */
#define SPL_FILE_OBJECT_READ_CSV           0x00000008 /* read via fgetcsv */
#define SPL_FILE_OBJECT_MASK               0x0000000F /* read via fgetcsv */

#define SPL_FILE_DIR_CURRENT_AS_FILEINFO   0x00000000 /* make RecursiveDirectoryTree::current() return SplFileInfo */
#define SPL_FILE_DIR_CURRENT_AS_SELF       0x00000010 /* make RecursiveDirectoryTree::current() return getSelf() */
#define SPL_FILE_DIR_CURRENT_AS_PATHNAME   0x00000020 /* make RecursiveDirectoryTree::current() return getPathname() */
#define SPL_FILE_DIR_CURRENT_MODE_MASK     0x000000F0 /* mask RecursiveDirectoryTree::current() */
#define SPL_FILE_DIR_CURRENT(intern,mode)  ((intern->flags&SPL_FILE_DIR_CURRENT_MODE_MASK)==mode)

#define SPL_FILE_DIR_KEY_AS_PATHNAME       0x00000000 /* make RecursiveDirectoryTree::key() return getPathname() */
#define SPL_FILE_DIR_KEY_AS_FILENAME       0x00000100 /* make RecursiveDirectoryTree::key() return getFilename() */
#define SPL_FILE_DIR_FOLLOW_SYMLINKS       0x00000200 /* make RecursiveDirectoryTree::hasChildren() follow symlinks */
#define SPL_FILE_DIR_KEY_MODE_MASK         0x00000F00 /* mask RecursiveDirectoryTree::key() */
#define SPL_FILE_DIR_KEY(intern,mode)      ((intern->flags&SPL_FILE_DIR_KEY_MODE_MASK)==mode)

#define SPL_FILE_DIR_SKIPDOTS              0x00001000 /* Tells whether it should skip dots or not */
#define SPL_FILE_DIR_UNIXPATHS             0x00002000 /* Whether to unixify path separators */
#define SPL_FILE_DIR_OTHERS_MASK           0x00003000 /* mask used for get/setFlags */

#endif /* SPL_DIRECTORY_H */

/*
 * Local Variables:
 * c-basic-offset: 4
 * tab-width: 4
 * End:
 * vim600: fdm=marker
 * vim: noet sw=4 ts=4
 */
```

```c
/*
  +----------------------------------------------------------------------+
  | PHP Version 7                                                        |
  +----------------------------------------------------------------------+
  | Copyright (c) 1997-2018 The PHP Group                                |
  +----------------------------------------------------------------------+
  | This source file is subject to version 3.01 of the PHP license,      |
  | that is bundled with this package in the file LICENSE, and is        |
  | available through the world-wide-web at the following url:           |
  | http://www.php.net/license/3_01.txt                                  |
  | If you did not receive a copy of the PHP license and are unable to   |
  | obtain it through the world-wide-web, please send a note to          |
  | license@php.net so we can mail you a copy immediately.               |
  +----------------------------------------------------------------------+
  | Author: Antony Dovgal <tony@daylessday.org>                          |
  |         Etienne Kneuss <colder@php.net>                              |
  +----------------------------------------------------------------------+
*/

/* $Id$ */

#ifdef HAVE_CONFIG_H
#include "config.h"
#endif

#include "php.h"
#include "php_ini.h"
#include "ext/standard/info.h"
#include "zend_exceptions.h"

#include "php_spl.h"
#include "spl_functions.h"
#include "spl_engine.h"
#include "spl_fixedarray.h"
#include "spl_exceptions.h"
#include "spl_iterators.h"

zend_object_handlers spl_handler_SplFixedArray;
PHPAPI zend_class_entry *spl_ce_SplFixedArray;

#ifdef COMPILE_DL_SPL_FIXEDARRAY
ZEND_GET_MODULE(spl_fixedarray)
#endif

typedef struct _spl_fixedarray { /* {{{ */
  zend_long size;
  zval *elements;
} spl_fixedarray;
/* }}} */

typedef struct _spl_fixedarray_object { /* {{{ */
  spl_fixedarray        array;
  zend_function         *fptr_offset_get;
  zend_function         *fptr_offset_set;
  zend_function         *fptr_offset_has;
  zend_function         *fptr_offset_del;
  zend_function         *fptr_count;
  int                    current;
  int                    flags;
  zend_class_entry      *ce_get_iterator;
  zend_object            std;
} spl_fixedarray_object;
/* }}} */

typedef struct _spl_fixedarray_it { /* {{{ */
  zend_user_iterator     intern;
} spl_fixedarray_it;
/* }}} */

#define SPL_FIXEDARRAY_OVERLOADED_REWIND  0x0001
#define SPL_FIXEDARRAY_OVERLOADED_VALID   0x0002
#define SPL_FIXEDARRAY_OVERLOADED_KEY     0x0004
#define SPL_FIXEDARRAY_OVERLOADED_CURRENT 0x0008
#define SPL_FIXEDARRAY_OVERLOADED_NEXT    0x0010

static inline spl_fixedarray_object *spl_fixed_array_from_obj(zend_object *obj) /* {{{ */ {
  return (spl_fixedarray_object*)((char*)(obj) - XtOffsetOf(spl_fixedarray_object, std));
}
/* }}} */

#define Z_SPLFIXEDARRAY_P(zv)  spl_fixed_array_from_obj(Z_OBJ_P((zv)))

static void spl_fixedarray_init(spl_fixedarray *array, zend_long size) /* {{{ */
{
  if (size > 0) {
    array->size = 0; /* reset size in case ecalloc() fails */
    array->elements = ecalloc(size, sizeof(zval));
    array->size = size;
  } else {
    array->elements = NULL;
    array->size = 0;
  }
}
/* }}} */

static void spl_fixedarray_resize(spl_fixedarray *array, zend_long size) /* {{{ */
{
  if (size == array->size) {
    /* nothing to do */
    return;
  }

  /* first initialization */
  if (array->size == 0) {
    spl_fixedarray_init(array, size);
    return;
  }

  /* clearing the array */
  if (size == 0) {
    zend_long i;

    for (i = 0; i < array->size; i++) {
      zval_ptr_dtor(&(array->elements[i]));
    }

    if (array->elements) {
      efree(array->elements);
      array->elements = NULL;
    }
  } else if (size > array->size) {
    array->elements = safe_erealloc(array->elements, size, sizeof(zval), 0);
    memset(array->elements + array->size, '\0', sizeof(zval) * (size - array->size));
  } else { /* size < array->size */
    zend_long i;

    for (i = size; i < array->size; i++) {
      zval_ptr_dtor(&(array->elements[i]));
    }
    array->elements = erealloc(array->elements, sizeof(zval) * size);
  }

  array->size = size;
}
/* }}} */

static void spl_fixedarray_copy(spl_fixedarray *to, spl_fixedarray *from) /* {{{ */
{
  int i;
  for (i = 0; i < from->size; i++) {
    ZVAL_COPY(&to->elements[i], &from->elements[i]);
  }
}
/* }}} */

static HashTable* spl_fixedarray_object_get_gc(zval *obj, zval **table, int *n) /* {{{{ */
{
  spl_fixedarray_object *intern  = Z_SPLFIXEDARRAY_P(obj);
  HashTable *ht = zend_std_get_properties(obj);

  *table = intern->array.elements;
  *n = (int)intern->array.size;

  return ht;
}
/* }}}} */

static HashTable* spl_fixedarray_object_get_properties(zval *obj) /* {{{{ */
{
  spl_fixedarray_object *intern  = Z_SPLFIXEDARRAY_P(obj);
  HashTable *ht = zend_std_get_properties(obj);
  zend_long  i = 0;

  if (intern->array.size > 0) {
    zend_long j = zend_hash_num_elements(ht);

    for (i = 0; i < intern->array.size; i++) {
      if (!Z_ISUNDEF(intern->array.elements[i])) {
        zend_hash_index_update(ht, i, &intern->array.elements[i]);
        Z_TRY_ADDREF(intern->array.elements[i]);
      } else {
        zend_hash_index_update(ht, i, &EG(uninitialized_zval));
      }
    }
    if (j > intern->array.size) {
      for (i = intern->array.size; i < j; ++i) {
        zend_hash_index_del(ht, i);
      }
    }
  }

  return ht;
}
/* }}}} */

static void spl_fixedarray_object_free_storage(zend_object *object) /* {{{ */
{
  spl_fixedarray_object *intern = spl_fixed_array_from_obj(object);
  zend_long i;

  if (intern->array.size > 0) {
    for (i = 0; i < intern->array.size; i++) {
      zval_ptr_dtor(&(intern->array.elements[i]));
    }

    if (intern->array.size > 0 && intern->array.elements) {
      efree(intern->array.elements);
    }
  }

  zend_object_std_dtor(&intern->std);
}
/* }}} */

zend_object_iterator *spl_fixedarray_get_iterator(zend_class_entry *ce, zval *object, int by_ref);

static zend_object *spl_fixedarray_object_new_ex(zend_class_entry *class_type, zval *orig, int clone_orig) /* {{{ */
{
  spl_fixedarray_object *intern;
  zend_class_entry     *parent = class_type;
  int                   inherited = 0;

  intern = zend_object_alloc(sizeof(spl_fixedarray_object), parent);

  zend_object_std_init(&intern->std, class_type);
  object_properties_init(&intern->std, class_type);

  intern->current = 0;
  intern->flags = 0;

  if (orig && clone_orig) {
    spl_fixedarray_object *other = Z_SPLFIXEDARRAY_P(orig);
    intern->ce_get_iterator = other->ce_get_iterator;
    spl_fixedarray_init(&intern->array, other->array.size);
    spl_fixedarray_copy(&intern->array, &other->array);
  }

  while (parent) {
    if (parent == spl_ce_SplFixedArray) {
      intern->std.handlers = &spl_handler_SplFixedArray;
      class_type->get_iterator = spl_fixedarray_get_iterator;
      break;
    }

    parent = parent->parent;
    inherited = 1;
  }

  if (!parent) { /* this must never happen */
    php_error_docref(NULL, E_COMPILE_ERROR, "Internal compiler error, Class is not child of SplFixedArray");
  }

  if (!class_type->iterator_funcs.zf_current) {
    class_type->iterator_funcs.zf_rewind = zend_hash_str_find_ptr(&class_type->function_table, "rewind", sizeof("rewind") - 1);
    class_type->iterator_funcs.zf_valid = zend_hash_str_find_ptr(&class_type->function_table, "valid", sizeof("valid") - 1);
    class_type->iterator_funcs.zf_key = zend_hash_str_find_ptr(&class_type->function_table, "key", sizeof("key") - 1);
    class_type->iterator_funcs.zf_current = zend_hash_str_find_ptr(&class_type->function_table, "current", sizeof("current") - 1);
    class_type->iterator_funcs.zf_next = zend_hash_str_find_ptr(&class_type->function_table, "next", sizeof("next") - 1);
  }
  if (inherited) {
    if (class_type->iterator_funcs.zf_rewind->common.scope  != parent) {
      intern->flags |= SPL_FIXEDARRAY_OVERLOADED_REWIND;
    }
    if (class_type->iterator_funcs.zf_valid->common.scope   != parent) {
      intern->flags |= SPL_FIXEDARRAY_OVERLOADED_VALID;
    }
    if (class_type->iterator_funcs.zf_key->common.scope     != parent) {
      intern->flags |= SPL_FIXEDARRAY_OVERLOADED_KEY;
    }
    if (class_type->iterator_funcs.zf_current->common.scope != parent) {
      intern->flags |= SPL_FIXEDARRAY_OVERLOADED_CURRENT;
    }
    if (class_type->iterator_funcs.zf_next->common.scope    != parent) {
      intern->flags |= SPL_FIXEDARRAY_OVERLOADED_NEXT;
    }

    intern->fptr_offset_get = zend_hash_str_find_ptr(&class_type->function_table, "offsetget", sizeof("offsetget") - 1);
    if (intern->fptr_offset_get->common.scope == parent) {
      intern->fptr_offset_get = NULL;
    }
    intern->fptr_offset_set = zend_hash_str_find_ptr(&class_type->function_table, "offsetset", sizeof("offsetset") - 1);
    if (intern->fptr_offset_set->common.scope == parent) {
      intern->fptr_offset_set = NULL;
    }
    intern->fptr_offset_has = zend_hash_str_find_ptr(&class_type->function_table, "offsetexists", sizeof("offsetexists") - 1);
    if (intern->fptr_offset_has->common.scope == parent) {
      intern->fptr_offset_has = NULL;
    }
    intern->fptr_offset_del = zend_hash_str_find_ptr(&class_type->function_table, "offsetunset", sizeof("offsetunset") - 1);
    if (intern->fptr_offset_del->common.scope == parent) {
      intern->fptr_offset_del = NULL;
    }
    intern->fptr_count = zend_hash_str_find_ptr(&class_type->function_table, "count", sizeof("count") - 1);
    if (intern->fptr_count->common.scope == parent) {
      intern->fptr_count = NULL;
    }
  }

  return &intern->std;
}
/* }}} */

static zend_object *spl_fixedarray_new(zend_class_entry *class_type) /* {{{ */
{
  return spl_fixedarray_object_new_ex(class_type, NULL, 0);
}
/* }}} */

static zend_object *spl_fixedarray_object_clone(zval *zobject) /* {{{ */
{
  zend_object *old_object;
  zend_object *new_object;

  old_object  = Z_OBJ_P(zobject);
  new_object = spl_fixedarray_object_new_ex(old_object->ce, zobject, 1);

  zend_objects_clone_members(new_object, old_object);

  return new_object;
}
/* }}} */

static inline zval *spl_fixedarray_object_read_dimension_helper(spl_fixedarray_object *intern, zval *offset) /* {{{ */
{
  zend_long index;

  /* we have to return NULL on error here to avoid memleak because of
   * ZE duplicating uninitialized_zval_ptr */
  if (!offset) {
    zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
    return NULL;
  }

  if (Z_TYPE_P(offset) != IS_LONG) {
    index = spl_offset_convert_to_long(offset);
  } else {
    index = Z_LVAL_P(offset);
  }

  if (index < 0 || index >= intern->array.size) {
    zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
    return NULL;
  } else if (Z_ISUNDEF(intern->array.elements[index])) {
    return NULL;
  } else {
    return &intern->array.elements[index];
  }
}
/* }}} */

static zval *spl_fixedarray_object_read_dimension(zval *object, zval *offset, int type, zval *rv) /* {{{ */
{
  spl_fixedarray_object *intern;

  intern = Z_SPLFIXEDARRAY_P(object);

  if (type == BP_VAR_IS && intern->fptr_offset_has) {
    SEPARATE_ARG_IF_REF(offset);
    zend_call_method_with_1_params(object, intern->std.ce, &intern->fptr_offset_has, "offsetexists", rv, offset);
    if (UNEXPECTED(Z_ISUNDEF_P(rv))) {
      zval_ptr_dtor(offset);
      return NULL;
    }
    if (!i_zend_is_true(rv)) {
      zval_ptr_dtor(offset);
      zval_ptr_dtor(rv);
      return &EG(uninitialized_zval);
    }
    zval_ptr_dtor(rv);
  }

  if (intern->fptr_offset_get) {
    zval tmp;
    if (!offset) {
      ZVAL_NULL(&tmp);
      offset = &tmp;
    } else {
      SEPARATE_ARG_IF_REF(offset);
    }
    zend_call_method_with_1_params(object, intern->std.ce, &intern->fptr_offset_get, "offsetGet", rv, offset);
    zval_ptr_dtor(offset);
    if (!Z_ISUNDEF_P(rv)) {
      return rv;
    }
    return &EG(uninitialized_zval);
  }

  return spl_fixedarray_object_read_dimension_helper(intern, offset);
}
/* }}} */

static inline void spl_fixedarray_object_write_dimension_helper(spl_fixedarray_object *intern, zval *offset, zval *value) /* {{{ */
{
  zend_long index;

  if (!offset) {
    /* '$array[] = value' syntax is not supported */
    zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
    return;
  }

  if (Z_TYPE_P(offset) != IS_LONG) {
    index = spl_offset_convert_to_long(offset);
  } else {
    index = Z_LVAL_P(offset);
  }

  if (index < 0 || index >= intern->array.size) {
    zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
    return;
  } else {
    if (!Z_ISUNDEF(intern->array.elements[index])) {
      zval_ptr_dtor(&(intern->array.elements[index]));
    }
    ZVAL_DEREF(value);
    ZVAL_COPY(&intern->array.elements[index], value);
  }
}
/* }}} */

static void spl_fixedarray_object_write_dimension(zval *object, zval *offset, zval *value) /* {{{ */
{
  spl_fixedarray_object *intern;
  zval tmp;

  intern = Z_SPLFIXEDARRAY_P(object);

  if (intern->fptr_offset_set) {
    if (!offset) {
      ZVAL_NULL(&tmp);
      offset = &tmp;
    } else {
      SEPARATE_ARG_IF_REF(offset);
    }
    SEPARATE_ARG_IF_REF(value);
    zend_call_method_with_2_params(object, intern->std.ce, &intern->fptr_offset_set, "offsetSet", NULL, offset, value);
    zval_ptr_dtor(value);
    zval_ptr_dtor(offset);
    return;
  }

  spl_fixedarray_object_write_dimension_helper(intern, offset, value);
}
/* }}} */

static inline void spl_fixedarray_object_unset_dimension_helper(spl_fixedarray_object *intern, zval *offset) /* {{{ */
{
  zend_long index;

  if (Z_TYPE_P(offset) != IS_LONG) {
    index = spl_offset_convert_to_long(offset);
  } else {
    index = Z_LVAL_P(offset);
  }

  if (index < 0 || index >= intern->array.size) {
    zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
    return;
  } else {
    zval_ptr_dtor(&(intern->array.elements[index]));
    ZVAL_UNDEF(&intern->array.elements[index]);
  }
}
/* }}} */

static void spl_fixedarray_object_unset_dimension(zval *object, zval *offset) /* {{{ */
{
  spl_fixedarray_object *intern;

  intern = Z_SPLFIXEDARRAY_P(object);

  if (intern->fptr_offset_del) {
    SEPARATE_ARG_IF_REF(offset);
    zend_call_method_with_1_params(object, intern->std.ce, &intern->fptr_offset_del, "offsetUnset", NULL, offset);
    zval_ptr_dtor(offset);
    return;
  }

  spl_fixedarray_object_unset_dimension_helper(intern, offset);

}
/* }}} */

static inline int spl_fixedarray_object_has_dimension_helper(spl_fixedarray_object *intern, zval *offset, int check_empty) /* {{{ */
{
  zend_long index;
  int retval;

  if (Z_TYPE_P(offset) != IS_LONG) {
    index = spl_offset_convert_to_long(offset);
  } else {
    index = Z_LVAL_P(offset);
  }

  if (index < 0 || index >= intern->array.size) {
    retval = 0;
  } else {
    if (Z_ISUNDEF(intern->array.elements[index])) {
      retval = 0;
    } else if (check_empty) {
      if (zend_is_true(&intern->array.elements[index])) {
        retval = 1;
      } else {
        retval = 0;
      }
    } else { /* != NULL and !check_empty */
      retval = 1;
    }
  }

  return retval;
}
/* }}} */

static int spl_fixedarray_object_has_dimension(zval *object, zval *offset, int check_empty) /* {{{ */
{
  spl_fixedarray_object *intern;

  intern = Z_SPLFIXEDARRAY_P(object);

  if (intern->fptr_offset_has) {
    zval rv;
    SEPARATE_ARG_IF_REF(offset);
    zend_call_method_with_1_params(object, intern->std.ce, &intern->fptr_offset_has, "offsetExists", &rv, offset);
    zval_ptr_dtor(offset);
    if (!Z_ISUNDEF(rv)) {
      zend_bool result = zend_is_true(&rv);
      zval_ptr_dtor(&rv);
      return result;
    }
    return 0;
  }

  return spl_fixedarray_object_has_dimension_helper(intern, offset, check_empty);
}
/* }}} */

static int spl_fixedarray_object_count_elements(zval *object, zend_long *count) /* {{{ */
{
  spl_fixedarray_object *intern;

  intern = Z_SPLFIXEDARRAY_P(object);
  if (intern->fptr_count) {
    zval rv;
    zend_call_method_with_0_params(object, intern->std.ce, &intern->fptr_count, "count", &rv);
    if (!Z_ISUNDEF(rv)) {
      *count = zval_get_long(&rv);
      zval_ptr_dtor(&rv);
    } else {
      *count = 0;
    }
  } else {
    *count = intern->array.size;
  }
  return SUCCESS;
}
/* }}} */

/* {{{ proto void SplFixedArray::__construct([int size])
*/
SPL_METHOD(SplFixedArray, __construct)
{
  zval *object = getThis();
  spl_fixedarray_object *intern;
  zend_long size = 0;

  if (zend_parse_parameters_throw(ZEND_NUM_ARGS(), "|l", &size) == FAILURE) {
    return;
  }

  if (size < 0) {
    zend_throw_exception_ex(spl_ce_InvalidArgumentException, 0, "array size cannot be less than zero");
    return;
  }

  intern = Z_SPLFIXEDARRAY_P(object);

  if (intern->array.size > 0) {
    /* called __construct() twice, bail out */
    return;
  }

  spl_fixedarray_init(&intern->array, size);
}
/* }}} */

/* {{{ proto void SplFixedArray::__wakeup()
*/
SPL_METHOD(SplFixedArray, __wakeup)
{
  spl_fixedarray_object *intern = Z_SPLFIXEDARRAY_P(getThis());
  HashTable *intern_ht = zend_std_get_properties(getThis());
  zval *data;

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  if (intern->array.size == 0) {
    int index = 0;
    int size = zend_hash_num_elements(intern_ht);

    spl_fixedarray_init(&intern->array, size);

    ZEND_HASH_FOREACH_VAL(intern_ht, data) {
      ZVAL_COPY(&intern->array.elements[index], data);
      index++;
    } ZEND_HASH_FOREACH_END();

    /* Remove the unserialised properties, since we now have the elements
     * within the spl_fixedarray_object structure. */
    zend_hash_clean(intern_ht);
  }
}
/* }}} */

/* {{{ proto int SplFixedArray::count(void)
*/
SPL_METHOD(SplFixedArray, count)
{
  zval *object = getThis();
  spl_fixedarray_object *intern;

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  intern = Z_SPLFIXEDARRAY_P(object);
  RETURN_LONG(intern->array.size);
}
/* }}} */

/* {{{ proto object SplFixedArray::toArray()
*/
SPL_METHOD(SplFixedArray, toArray)
{
  spl_fixedarray_object *intern;

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  intern = Z_SPLFIXEDARRAY_P(getThis());

  if (intern->array.size > 0) {
    int i = 0;

    array_init(return_value);
    for (; i < intern->array.size; i++) {
      if (!Z_ISUNDEF(intern->array.elements[i])) {
        zend_hash_index_update(Z_ARRVAL_P(return_value), i, &intern->array.elements[i]);
        Z_TRY_ADDREF(intern->array.elements[i]);
      } else {
        zend_hash_index_update(Z_ARRVAL_P(return_value), i, &EG(uninitialized_zval));
      }
    }
  } else {
    ZVAL_EMPTY_ARRAY(return_value);
  }
}
/* }}} */

/* {{{ proto object SplFixedArray::fromArray(array data[, bool save_indexes])
*/
SPL_METHOD(SplFixedArray, fromArray)
{
  zval *data;
  spl_fixedarray array;
  spl_fixedarray_object *intern;
  int num;
  zend_bool save_indexes = 1;

  if (zend_parse_parameters(ZEND_NUM_ARGS(), "a|b", &data, &save_indexes) == FAILURE) {
    return;
  }

  num = zend_hash_num_elements(Z_ARRVAL_P(data));

  if (num > 0 && save_indexes) {
    zval *element;
    zend_string *str_index;
    zend_ulong num_index, max_index = 0;
    zend_long tmp;

    ZEND_HASH_FOREACH_KEY(Z_ARRVAL_P(data), num_index, str_index) {
      if (str_index != NULL || (zend_long)num_index < 0) {
        zend_throw_exception_ex(spl_ce_InvalidArgumentException, 0, "array must contain only positive integer keys");
        return;
      }

      if (num_index > max_index) {
        max_index = num_index;
      }
    } ZEND_HASH_FOREACH_END();

    tmp = max_index + 1;
    if (tmp <= 0) {
      zend_throw_exception_ex(spl_ce_InvalidArgumentException, 0, "integer overflow detected");
      return;
    }
    spl_fixedarray_init(&array, tmp);

    ZEND_HASH_FOREACH_KEY_VAL(Z_ARRVAL_P(data), num_index, str_index, element) {
      ZVAL_DEREF(element);
      ZVAL_COPY(&array.elements[num_index], element);
    } ZEND_HASH_FOREACH_END();

  } else if (num > 0 && !save_indexes) {
    zval *element;
    zend_long i = 0;

    spl_fixedarray_init(&array, num);

    ZEND_HASH_FOREACH_VAL(Z_ARRVAL_P(data), element) {
      ZVAL_DEREF(element);
      ZVAL_COPY(&array.elements[i], element);
      i++;
    } ZEND_HASH_FOREACH_END();
  } else {
    spl_fixedarray_init(&array, 0);
  }

  object_init_ex(return_value, spl_ce_SplFixedArray);

  intern = Z_SPLFIXEDARRAY_P(return_value);
  intern->array = array;
}
/* }}} */

/* {{{ proto int SplFixedArray::getSize(void)
*/
SPL_METHOD(SplFixedArray, getSize)
{
  zval *object = getThis();
  spl_fixedarray_object *intern;

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  intern = Z_SPLFIXEDARRAY_P(object);
  RETURN_LONG(intern->array.size);
}
/* }}} */

/* {{{ proto bool SplFixedArray::setSize(int size)
*/
SPL_METHOD(SplFixedArray, setSize)
{
  zval *object = getThis();
  spl_fixedarray_object *intern;
  zend_long size;

  if (zend_parse_parameters(ZEND_NUM_ARGS(), "l", &size) == FAILURE) {
    return;
  }

  if (size < 0) {
    zend_throw_exception_ex(spl_ce_InvalidArgumentException, 0, "array size cannot be less than zero");
    return;
  }

  intern = Z_SPLFIXEDARRAY_P(object);

  spl_fixedarray_resize(&intern->array, size);
  RETURN_TRUE;
}
/* }}} */

/* {{{ proto bool SplFixedArray::offsetExists(mixed $index)
 Returns whether the requested $index exists. */
SPL_METHOD(SplFixedArray, offsetExists)
{
  zval                  *zindex;
  spl_fixedarray_object  *intern;

  if (zend_parse_parameters(ZEND_NUM_ARGS(), "z", &zindex) == FAILURE) {
    return;
  }

  intern = Z_SPLFIXEDARRAY_P(getThis());

  RETURN_BOOL(spl_fixedarray_object_has_dimension_helper(intern, zindex, 0));
} /* }}} */

/* {{{ proto mixed SplFixedArray::offsetGet(mixed $index)
 Returns the value at the specified $index. */
SPL_METHOD(SplFixedArray, offsetGet)
{
  zval *zindex, *value;
  spl_fixedarray_object  *intern;

  if (zend_parse_parameters(ZEND_NUM_ARGS(), "z", &zindex) == FAILURE) {
    return;
  }

  intern = Z_SPLFIXEDARRAY_P(getThis());
  value = spl_fixedarray_object_read_dimension_helper(intern, zindex);

  if (value) {
    ZVAL_DEREF(value);
    ZVAL_COPY(return_value, value);
  } else {
    RETURN_NULL();
  }
} /* }}} */

/* {{{ proto void SplFixedArray::offsetSet(mixed $index, mixed $newval)
 Sets the value at the specified $index to $newval. */
SPL_METHOD(SplFixedArray, offsetSet)
{
  zval                  *zindex, *value;
  spl_fixedarray_object  *intern;

  if (zend_parse_parameters(ZEND_NUM_ARGS(), "zz", &zindex, &value) == FAILURE) {
    return;
  }

  intern = Z_SPLFIXEDARRAY_P(getThis());
  spl_fixedarray_object_write_dimension_helper(intern, zindex, value);

} /* }}} */

/* {{{ proto void SplFixedArray::offsetUnset(mixed $index)
 Unsets the value at the specified $index. */
SPL_METHOD(SplFixedArray, offsetUnset)
{
  zval                  *zindex;
  spl_fixedarray_object  *intern;

  if (zend_parse_parameters(ZEND_NUM_ARGS(), "z", &zindex) == FAILURE) {
    return;
  }

  intern = Z_SPLFIXEDARRAY_P(getThis());
  spl_fixedarray_object_unset_dimension_helper(intern, zindex);

} /* }}} */

static void spl_fixedarray_it_dtor(zend_object_iterator *iter) /* {{{ */
{
  spl_fixedarray_it  *iterator = (spl_fixedarray_it *)iter;

  zend_user_it_invalidate_current(iter);
  zval_ptr_dtor(&iterator->intern.it.data);
}
/* }}} */

static void spl_fixedarray_it_rewind(zend_object_iterator *iter) /* {{{ */
{
  spl_fixedarray_object *object = Z_SPLFIXEDARRAY_P(&iter->data);

  if (object->flags & SPL_FIXEDARRAY_OVERLOADED_REWIND) {
    zend_user_it_rewind(iter);
  } else {
    object->current = 0;
  }
}
/* }}} */

static int spl_fixedarray_it_valid(zend_object_iterator *iter) /* {{{ */
{
  spl_fixedarray_object *object = Z_SPLFIXEDARRAY_P(&iter->data);

  if (object->flags & SPL_FIXEDARRAY_OVERLOADED_VALID) {
    return zend_user_it_valid(iter);
  }

  if (object->current >= 0 && object->current < object->array.size) {
    return SUCCESS;
  }

  return FAILURE;
}
/* }}} */

static zval *spl_fixedarray_it_get_current_data(zend_object_iterator *iter) /* {{{ */
{
  zval zindex;
  spl_fixedarray_object *object = Z_SPLFIXEDARRAY_P(&iter->data);

  if (object->flags & SPL_FIXEDARRAY_OVERLOADED_CURRENT) {
    return zend_user_it_get_current_data(iter);
  } else {
    zval *data;

    ZVAL_LONG(&zindex, object->current);

    data = spl_fixedarray_object_read_dimension_helper(object, &zindex);
    zval_ptr_dtor(&zindex);

    if (data == NULL) {
      data = &EG(uninitialized_zval);
    }
    return data;
  }
}
/* }}} */

static void spl_fixedarray_it_get_current_key(zend_object_iterator *iter, zval *key) /* {{{ */
{
  spl_fixedarray_object *object = Z_SPLFIXEDARRAY_P(&iter->data);

  if (object->flags & SPL_FIXEDARRAY_OVERLOADED_KEY) {
    zend_user_it_get_current_key(iter, key);
  } else {
    ZVAL_LONG(key, object->current);
  }
}
/* }}} */

static void spl_fixedarray_it_move_forward(zend_object_iterator *iter) /* {{{ */
{
  spl_fixedarray_object *object = Z_SPLFIXEDARRAY_P(&iter->data);

  if (object->flags & SPL_FIXEDARRAY_OVERLOADED_NEXT) {
    zend_user_it_move_forward(iter);
  } else {
    zend_user_it_invalidate_current(iter);
    object->current++;
  }
}
/* }}} */

/* {{{  proto int SplFixedArray::key()
   Return current array key */
SPL_METHOD(SplFixedArray, key)
{
  spl_fixedarray_object *intern = Z_SPLFIXEDARRAY_P(getThis());

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  RETURN_LONG(intern->current);
}
/* }}} */

/* {{{ proto void SplFixedArray::next()
   Move to next entry */
SPL_METHOD(SplFixedArray, next)
{
  spl_fixedarray_object *intern = Z_SPLFIXEDARRAY_P(getThis());

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  intern->current++;
}
/* }}} */

/* {{{ proto bool SplFixedArray::valid()
   Check whether the datastructure contains more entries */
SPL_METHOD(SplFixedArray, valid)
{
  spl_fixedarray_object *intern = Z_SPLFIXEDARRAY_P(getThis());

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  RETURN_BOOL(intern->current >= 0 && intern->current < intern->array.size);
}
/* }}} */

/* {{{ proto void SplFixedArray::rewind()
   Rewind the datastructure back to the start */
SPL_METHOD(SplFixedArray, rewind)
{
  spl_fixedarray_object *intern = Z_SPLFIXEDARRAY_P(getThis());

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  intern->current = 0;
}
/* }}} */

/* {{{ proto mixed|NULL SplFixedArray::current()
   Return current datastructure entry */
SPL_METHOD(SplFixedArray, current)
{
  zval zindex, *value;
  spl_fixedarray_object *intern  = Z_SPLFIXEDARRAY_P(getThis());

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  ZVAL_LONG(&zindex, intern->current);

  value = spl_fixedarray_object_read_dimension_helper(intern, &zindex);

  zval_ptr_dtor(&zindex);

  if (value) {
    ZVAL_DEREF(value);
    ZVAL_COPY(return_value, value);
  } else {
    RETURN_NULL();
  }
}
/* }}} */

/* iterator handler table */
static const zend_object_iterator_funcs spl_fixedarray_it_funcs = {
  spl_fixedarray_it_dtor,
  spl_fixedarray_it_valid,
  spl_fixedarray_it_get_current_data,
  spl_fixedarray_it_get_current_key,
  spl_fixedarray_it_move_forward,
  spl_fixedarray_it_rewind,
  NULL
};

zend_object_iterator *spl_fixedarray_get_iterator(zend_class_entry *ce, zval *object, int by_ref) /* {{{ */
{
  spl_fixedarray_it *iterator;

  if (by_ref) {
    zend_throw_exception(spl_ce_RuntimeException, "An iterator cannot be used with foreach by reference", 0);
    return NULL;
  }

  iterator = emalloc(sizeof(spl_fixedarray_it));

  zend_iterator_init((zend_object_iterator*)iterator);

  ZVAL_COPY(&iterator->intern.it.data, object);
  iterator->intern.it.funcs = &spl_fixedarray_it_funcs;
  iterator->intern.ce = ce;
  ZVAL_UNDEF(&iterator->intern.value);

  return &iterator->intern.it;
}
/* }}} */

ZEND_BEGIN_ARG_INFO_EX(arginfo_splfixedarray_construct, 0, 0, 0)
  ZEND_ARG_INFO(0, size)
ZEND_END_ARG_INFO()

ZEND_BEGIN_ARG_INFO_EX(arginfo_fixedarray_offsetGet, 0, 0, 1)
  ZEND_ARG_INFO(0, index)
ZEND_END_ARG_INFO()

ZEND_BEGIN_ARG_INFO_EX(arginfo_fixedarray_offsetSet, 0, 0, 2)
  ZEND_ARG_INFO(0, index)
  ZEND_ARG_INFO(0, newval)
ZEND_END_ARG_INFO()

ZEND_BEGIN_ARG_INFO(arginfo_fixedarray_setSize, 0)
  ZEND_ARG_INFO(0, value)
ZEND_END_ARG_INFO()

ZEND_BEGIN_ARG_INFO_EX(arginfo_fixedarray_fromArray, 0, 0, 1)
  ZEND_ARG_INFO(0, data)
  ZEND_ARG_INFO(0, save_indexes)
ZEND_END_ARG_INFO()

ZEND_BEGIN_ARG_INFO(arginfo_splfixedarray_void, 0)
ZEND_END_ARG_INFO()

static const zend_function_entry spl_funcs_SplFixedArray[] = { /* {{{ */
  SPL_ME(SplFixedArray, __construct,     arginfo_splfixedarray_construct,ZEND_ACC_PUBLIC)
  SPL_ME(SplFixedArray, __wakeup,        arginfo_splfixedarray_void,     ZEND_ACC_PUBLIC)
  SPL_ME(SplFixedArray, count,           arginfo_splfixedarray_void,     ZEND_ACC_PUBLIC)
  SPL_ME(SplFixedArray, toArray,         arginfo_splfixedarray_void,     ZEND_ACC_PUBLIC)
  SPL_ME(SplFixedArray, fromArray,       arginfo_fixedarray_fromArray,   ZEND_ACC_PUBLIC|ZEND_ACC_STATIC)
  SPL_ME(SplFixedArray, getSize,         arginfo_splfixedarray_void,     ZEND_ACC_PUBLIC)
  SPL_ME(SplFixedArray, setSize,         arginfo_fixedarray_setSize,     ZEND_ACC_PUBLIC)
  SPL_ME(SplFixedArray, offsetExists,    arginfo_fixedarray_offsetGet,   ZEND_ACC_PUBLIC)
  SPL_ME(SplFixedArray, offsetGet,       arginfo_fixedarray_offsetGet,   ZEND_ACC_PUBLIC)
  SPL_ME(SplFixedArray, offsetSet,       arginfo_fixedarray_offsetSet,   ZEND_ACC_PUBLIC)
  SPL_ME(SplFixedArray, offsetUnset,     arginfo_fixedarray_offsetGet,   ZEND_ACC_PUBLIC)
  SPL_ME(SplFixedArray, rewind,          arginfo_splfixedarray_void,     ZEND_ACC_PUBLIC)
  SPL_ME(SplFixedArray, current,         arginfo_splfixedarray_void,     ZEND_ACC_PUBLIC)
  SPL_ME(SplFixedArray, key,             arginfo_splfixedarray_void,     ZEND_ACC_PUBLIC)
  SPL_ME(SplFixedArray, next,            arginfo_splfixedarray_void,     ZEND_ACC_PUBLIC)
  SPL_ME(SplFixedArray, valid,           arginfo_splfixedarray_void,     ZEND_ACC_PUBLIC)
  PHP_FE_END
};
/* }}} */

/* {{{ PHP_MINIT_FUNCTION */
PHP_MINIT_FUNCTION(spl_fixedarray)
{
  REGISTER_SPL_STD_CLASS_EX(SplFixedArray, spl_fixedarray_new, spl_funcs_SplFixedArray);
  memcpy(&spl_handler_SplFixedArray, zend_get_std_object_handlers(), sizeof(zend_object_handlers));

  spl_handler_SplFixedArray.offset          = XtOffsetOf(spl_fixedarray_object, std);
  spl_handler_SplFixedArray.clone_obj       = spl_fixedarray_object_clone;
  spl_handler_SplFixedArray.read_dimension  = spl_fixedarray_object_read_dimension;
  spl_handler_SplFixedArray.write_dimension = spl_fixedarray_object_write_dimension;
  spl_handler_SplFixedArray.unset_dimension = spl_fixedarray_object_unset_dimension;
  spl_handler_SplFixedArray.has_dimension   = spl_fixedarray_object_has_dimension;
  spl_handler_SplFixedArray.count_elements  = spl_fixedarray_object_count_elements;
  spl_handler_SplFixedArray.get_properties  = spl_fixedarray_object_get_properties;
  spl_handler_SplFixedArray.get_gc          = spl_fixedarray_object_get_gc;
  spl_handler_SplFixedArray.dtor_obj        = zend_objects_destroy_object;
  spl_handler_SplFixedArray.free_obj        = spl_fixedarray_object_free_storage;

  REGISTER_SPL_IMPLEMENTS(SplFixedArray, Iterator);
  REGISTER_SPL_IMPLEMENTS(SplFixedArray, ArrayAccess);
  REGISTER_SPL_IMPLEMENTS(SplFixedArray, Countable);

  spl_ce_SplFixedArray->get_iterator = spl_fixedarray_get_iterator;

  return SUCCESS;
}
/* }}} */


/*
 * Local variables:
 * tab-width: 4
 * c-basic-offset: 4
 * End:
 * vim600: noet sw=4 ts=4 fdm=marker
 * vim<600: noet sw=4 ts=4
 */
```

```
/*
   +----------------------------------------------------------------------+
   | PHP Version 7                                                        |
   +----------------------------------------------------------------------+
   | Copyright (c) 1997-2018 The PHP Group                                |
   +----------------------------------------------------------------------+
   | This source file is subject to version 3.01 of the PHP license,      |
   | that is bundled with this package in the file LICENSE, and is        |
   | available through the world-wide-web at the following url:           |
   | http://www.php.net/license/3_01.txt                                  |
   | If you did not receive a copy of the PHP license and are unable to   |
   | obtain it through the world-wide-web, please send a note to          |
   | license@php.net so we can mail you a copy immediately.               |
   +----------------------------------------------------------------------+
   | Author: Antony Dovgal <tony@daylessday.org>                          |
   |         Etienne Kneuss <colder@php.net>                              |
   +----------------------------------------------------------------------+
*/

/* $Id$ */

#ifndef SPL_FIXEDARRAY_H
#define SPL_FIXEDARRAY_H

extern PHPAPI zend_class_entry *spl_ce_SplFixedArray;

PHP_MINIT_FUNCTION(spl_fixedarray);

#endif  /* SPL_FIXEDARRAY_H */

/*
 * Local variables:
 * tab-width: 4
 * c-basic-offset: 4
 * End:
 * vim600: noet sw=4 ts=4 fdm=marker
 * vim<600: noet sw=4 ts=4
 */
```

```c
/*
   +----------------------------------------------------------------------+
   | PHP Version 7                                                        |
   +----------------------------------------------------------------------+
   | Copyright (c) 1997-2018 The PHP Group                                |
   +----------------------------------------------------------------------+
   | This source file is subject to version 3.01 of the PHP license,      |
   | that is bundled with this package in the file LICENSE, and is        |
   | available through the world-wide-web at the following url:           |
   | http://www.php.net/license/3_01.txt                                  |
   | If you did not receive a copy of the PHP license and are unable to   |
   | obtain it through the world-wide-web, please send a note to          |
   | license@php.net so we can mail you a copy immediately.               |
   +----------------------------------------------------------------------+
   | Authors: Marcus Boerger <helly@php.net>                              |
   +----------------------------------------------------------------------+
 */

#ifndef PHP_SPL_H
#define PHP_SPL_H

#include "php.h"
#include <stdarg.h>

#define PHP_SPL_VERSION PHP_VERSION

extern zend_module_entry spl_module_entry;
#define phpext_spl_ptr &spl_module_entry

#ifdef PHP_WIN32
# ifdef SPL_EXPORTS
#   define SPL_API __declspec(dllexport)
# elif defined(COMPILE_DL_SPL)
#   define SPL_API __declspec(dllimport)
# else
#   define SPL_API /* nothing */
# endif
#elif defined(__GNUC__) && __GNUC__ >= 4
# define SPL_API __attribute__ ((visibility("default")))
#else
# define SPL_API
#endif

#if defined(PHP_WIN32) && !defined(COMPILE_DL_SPL)
#undef phpext_spl
#define phpext_spl NULL
#endif

PHP_MINIT_FUNCTION(spl);
PHP_MSHUTDOWN_FUNCTION(spl);
PHP_RINIT_FUNCTION(spl);
PHP_RSHUTDOWN_FUNCTION(spl);
PHP_MINFO_FUNCTION(spl);


ZEND_BEGIN_MODULE_GLOBALS(spl)
  zend_string *autoload_extensions;
  HashTable   *autoload_functions;
  intptr_t     hash_mask_handle;
  intptr_t     hash_mask_handlers;
  int          hash_mask_init;
  int          autoload_running;
ZEND_END_MODULE_GLOBALS(spl)

ZEND_EXTERN_MODULE_GLOBALS(spl)
#define SPL_G(v) ZEND_MODULE_GLOBALS_ACCESSOR(spl, v)

PHP_FUNCTION(spl_classes);
PHP_FUNCTION(class_parents);
PHP_FUNCTION(class_implements);
PHP_FUNCTION(class_uses);

PHPAPI zend_string *php_spl_object_hash(zval *obj);

#endif /* PHP_SPL_H */

/*
 * Local Variables:
 * c-basic-offset: 4
 * tab-width: 4
 * End:
 * vim600: fdm=marker
 * vim: noet sw=4 ts=4
 */
```